TRANSPARENT DISTRIBUTION FOR JAVA APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Philip McGachey

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2010

Purdue University

West Lafayette, Indiana

To all those who have supported me over the years.

ACKNOWLEDGMENTS

I'd first like to thank my PhD committee: Jan Vitek and Patrick Eugster for agreeing to serve, and Eliot Moss for his input both at the start and the end of the process. I particularly thank my advisor, Tony Hosking, for his years of guidance and support; during my career he has acted as a mentor, a colleague, a boss and a friend.

A great many people have helped and influenced me over the years at graduate school, making any list necessarily incomplete. I'd like to thank Greg Wright at Sun and Rick Hudson at Intel for supervising me during my various internships and giving me a valuable insight into life beyond academia. While at Purdue, I maintained my sanity largely with the help of Michael Richmond, Darrin and Karen Cox, Mike Steinhour, Joe Auffermann, Samuel Adams and the second Earl Grey. I'd particularly like to thank Dennis and Petra Brylow for their encouragement, help and bad TV, Adam Welc for his constant friendship, advice and beer, and Nicki Barker for her kindness and moral support during the last year of my degree.

I'd finally like to thank my parents for their unwavering support over what turned out to be a long process. Their enthusiasm and continual encouragement through the good times and bad gave me the confidence to see things through to the end.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

McGachey, Philip Ph.D., Purdue University, May 2010. Transparent Distribution for Java Applications. Major Professor: Antony L. Hosking.

Cloud computing and falling hardware prices today offer unprecedented access to cheap and flexible computer clusters. Unfortunately, developing the distributed applications that are needed to take full advantage of this extra capacity is still a daunting task. Application programmers must concern themselves not only with application logic, but also with the mechanics of distribution: tracking remote data, global synchronization, network configuration and so forth.

RuggedJ is a transparent Java distribution framework that relieves much of the burden from distributed programming. We inject distribution logic into standard Java applications, and we deploy the rewritten code across a dynamic run-time infrastructure. This way, we maintain the semantics of the original application while providing powerful distribution capabilities such as dynamic application partitioning, object location transparency and replication of immutable state.

This dissertation describes the design and implementation of our prototype RuggedJ transparent distribution infrastructure. It discusses both the bytecode rewriting techniques that allow us to distribute code and the run-time infrastructure that manages the distributed application. We investigate the properties of *distributable* applications (those that benefit from distribution), and describe techniques to optimize performance for the RuggedJ platform. Finally, we demonstrate that the system can distribute several realistic applications, and show that these applications exhibit scalability when running on a cluster beyond that possible on a single machine.

1 INTRODUCTION

Transparent distribution can allow distributable standard Java applications to execute across multiple machines with minimal programmer overhead. Transformed applications can show minimal performance degradation on a single host, while demonstrating significantly improved performance on a cluster.

With the increasing availability of affordable commodity clusters and cloud computing infrastructures, consumers have access to more computing power than ever before. Data sets and workloads are increasing to employ these resources; the growing internet economy requires that servers scale to handle ever-increasing levels of traffic, while scientific infrastructure produces massive amounts of raw data to be processed. The result is that developers today can no longer rely on *vertical* scaling, waiting for the rising tide of processor speed to improve single-threaded application performance. Rather, applications must scale *horizontally* by distributing across multiple loosely-connected computing resources and so leveraging the available computing capacity.

Developing applications that effectively take advantage of this capacity requires significant programmer effort. Systems must be designed to distribute across multiple machines, transferring data and accessing remote objects over network connections. Using commodity frameworks such as Java Remote Method Invocation (RMI) and serialization, programmers must explicitly track objects across disparate memory spaces, handle remote objects specially, and ensure that data is correctly synchronized. Further, obtaining good performance often requires implementing unsupported functionality such as object migration or replication to minimize remote invocations.

Transparent distribution removes much of this overhead from developers. Rather than writing an application with explicit distribution code, programmers use the familiar shared-

memory abstraction to develop *distributable* multi-threaded systems. The transparent distribution framework takes this application and dynamically inserts distribution logic, transforming the code at run time to run across multiple machines. A run-time system operates on each machine and maintains mappings between remote objects.

Such a system offers substantial advantages over manual distribution. The programming model is significantly simpler; developers do not need to explicitly handle the mechanisms of distributed execution. The distribution infrastructure can analyze the application to optimize object placement, replicating immutable data. Further, it can provide means by which objects can migrate from one machine to another, taking advantage of shifting data access patterns. Finally the developer does not have to find and track remote objects or global data, as the run-time system maintains this information.

A transparent distribution system requires several features to appeal to developers. It must offer significant savings in development effort when building distributed applications, allowing developers to focus on the specific functionality of their systems rather than the mechanism by which they are distributed. Additionally, the distributed application should not be tied to a specific network configuration; an application that scales horizontally must be able to deploy on an arbitrary network that may change between invocations. Finally, the system must not be a source of significant performance degradation, and must not present a barrier to the application's scaling across multiple machines.

1.1 Overview

We have developed a transparent distribution system for Java that meets these criteria. The RuggedJ system takes bytecode emitted from any standard Java compiler and uses a rewriting class loader to transform the application's code. The rewritten code integrates with a distributed run-time system that manages the global execution of the application. We have implemented and evaluated a prototype RuggedJ system, and have found that certain classes of applications can show significant performance improvements when run across multiple machines.

The dissertation is structured as follows:

- The remainder of this Introduction chapter gives an overview of the RuggedJ system, and enumerates the contributions of this work.
- Chapter 2 discusses the prior work in this area, and summarizes some closely related fields.
- Chapter 3 details the bytecode transformations by which we rewrite the classes of an application. This includes a comprehensive discussion of how we integrate Java library classes into our system.
- Chapter 4 outlines the run-time management system that oversees the execution of a distributed application. We first discuss some of the features of this infrastructure, such as remote object tracking and data replication. We then indicate how we build upon these underlying systems to support some of Java's more complex semantics.
- Chapter 5 provides a full discussion of distributable applications in Java, including those data structures and design features that lend themselves to distribution. We discuss a number of realistic applications that we have distributed using our prototype system, detailing the program structure and distribution strategies for each, as well as performance evaluations where appropriate.
- Finally, Chapter 6 summarizes the dissertation, and indicates some areas in which this work could be continued in the future.

1.2 RuggedJ

In this section we introduce RuggedJ, a specification-based, transparent distribution framework for Java. RuggedJ automatically (based on concise programmer specifications) transforms standard multi-threaded Java applications to run across a cluster of unmodified Java virtual machines (JVMs), removing the burden of explicit distributed programming from the developer. Our framework gives Java developers access to powerful distribution

mechanisms for minimal additional effort, while maintaining the semantics of the original application.

RuggedJ offers significant novel benefits. It allows distributed applications to scale out to clusters of arbitrary size and configuration without additional developer effort, and without hard-encoding the network and its topology within the application. This simplifies development and maintenance. Further, the developer need not worry about objects that may be local under some network configurations and remote under others, and need not explicitly handle references to migrating objects. Finally, RuggedJ transforms standard Java applications, with no additional annotations or dependencies upon specialized libraries.

1.2.1 Target Applications

RuggedJ can transform and distribute most standard Java applications (with minor exceptions; we do not support some Java features such as user-level class loading and certain aspects of reflection). However not all Java programs benefit from distribution. We target a class of applications that we refer to as *distributable*: those that can be broken into multiple discrete units, each of which is operated upon by a dedicated node. These *distribution units* are located on different nodes in the network, allowing scalability by adding additional nodes to operate over more units. To maximize performance, there should be little cross-talk between multiple distribution units. Interactions between distribution units on different nodes are performed via remote method invocation which carries with it an inherent performance penalty.

In addition to these properties, distributable applications should be long-running. Establishing a RuggedJ network incurs a startup cost that must be amortized across the execution of the application. Additionally, there is some work required to create threads and copy initial data. Thus, small applications (having execution times less than two or three minutes) will never see the benefits of distribution.

Finally, distributable applications should minimize their dependence on native code, reflection and static data. The presence of native or reflective code can tie the instances of a given class to a single node in the network, destroying any potential distribution (an issue that is discussed in depth in Chapter 3). Static data generally does not limit the distributability of an application, but can lead to increased network traffic, since static data must be globally unique within the system.

We discuss the characteristics of distributable applications more fully in Chapter 5.

1.2.2 System Design

RuggedJ runs across an interconnected network of JVMs that we refer to as *nodes*. Each physical machine (a *host*) within the cluster can contain one or more nodes. RuggedJ is implemented entirely at the bytecode level (we do not modify the JVM), and we make no requirements on the capabilities of the hosts, save that they each run a fully-featured Java virtual machine with compatible versions of the Java class libraries. This VM-agnosticism provides us with several advantages. First, our implementation is not tied to a specific VM distribution, making porting between VMs and versions of the Java specification trivial. Additionally, a given RuggedJ network can integrate heterogeneous hosts; an application can be distributed across multiple architectures running diverse operating systems, so long as they provide a compatible JVM. Figure 1.1 shows the construction of a RuggedJ network.

RuggedJ is designed to allow maximum flexibility in the platform upon which it runs. The *partitioning* of an application (the allocation of the application's data and work across multiple hosts) is determined at run-time, so an application need not be modified to account for every new network upon which it runs. This is achieved through our partitioning plug-in system: when targeting RuggedJ for distribution, the application developer supplies a partitioning strategy, encapsulated as a Java class. This class has access to the run-time state of the RuggedJ network and, guided by this information, indicates how the application

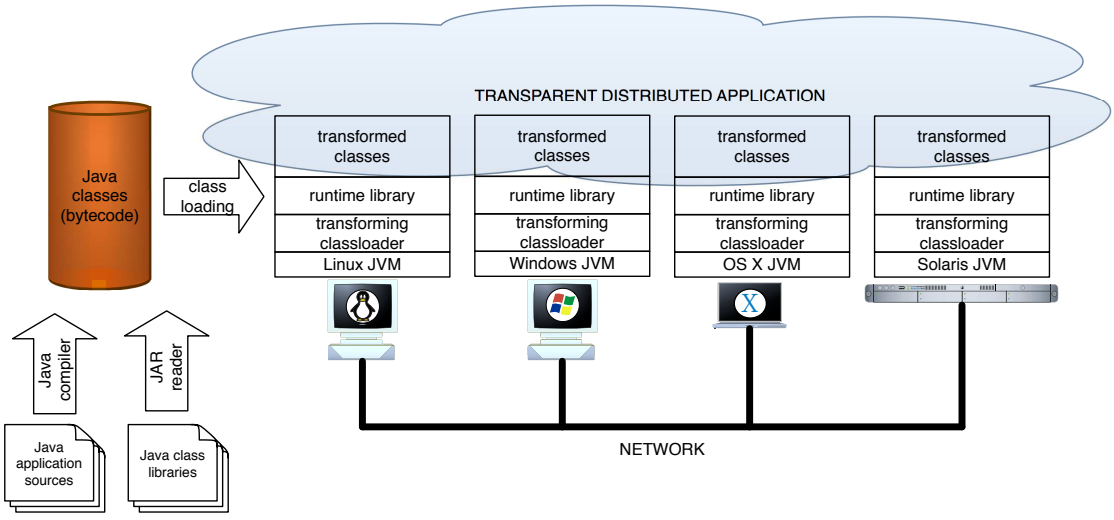


Figure 1.1.: The RuggedJ System Architecture

should be partitioned, and makes dynamic decisions about allocation, migration and so forth.

Each node in the RuggedJ system comprises two major components: a bytecode-rewriting class loader, and a run-time library. Classes loaded into the JVM are first processed by our class loader to inject distribution logic. These rewritten classes then interact with the run-time library on their local node. The run-time library itself manages the higher-level aspects of distribution. Among other functions, it tracks remote objects, handles threading and locking, maintains static data and coordinates remote method invocations. The run-time libraries of each node within the RuggedJ network interact with one another to coordinate global data, monitor the state of the network and propagate dynamic object information.

One node in the RuggedJ network is designated as the *head node*. This node performs some globally unique tasks in addition to its responsibilities as a standard node. For example, it invokes the `main` method that launches the application, it acts as a canonical source for network information (such as the locations of shared objects), and it maintains a terminal connection for the standard input, output and error streams. The head node is specified as part of the network configuration. The remaining nodes automatically organize them-

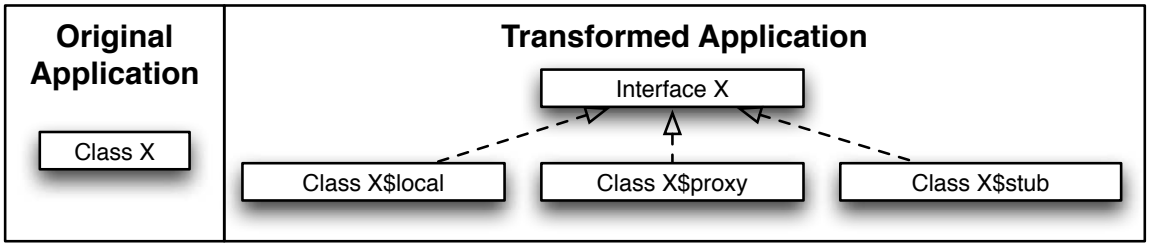


Figure 1.2.: The RuggedJ object model

selves into a tree hierarchy (with the head node at the root), allowing efficient broadcasting of data and a simple lookup mechanism for network state.

1.2.3 Class Transformation

Key to distribution of applications within RuggedJ is the bytecode transformation library. RuggedJ transforms the classes that make up an application to implement its object model [McGachey et al., 2009a], as shown in Figure 1.2. Each class from the original application spawns creation of three new classes and one interface. The interface represents the class's *protocol*: the original method names and signatures, and additional get/set methods for every field in the class. Object references within RuggedJ are typed exclusively by interface; abstracting out the protocol allows the concrete implementation of a class to vary without altering client code that refers to instances of the class. We rewrite method bodies within the class to refer to transformed objects, including redirecting method and field accesses through interfaces, modifying the types of objects to account for the object model, and so on.

The three classes `X_local`, `X_stub` and `X_proxy` provide these concrete implementations. A *local* object contains the fields and method implementations of the original class; it can be thought of as the canonical representation of the object. Local objects have a $1:1$ relationship with objects in the original application, so only one local object exists in the RuggedJ system for every object in the original application. The second class, the *stub*, implements the interface by providing remote method calls to the local object. Stubs have an

$n : 1$ relationship to local objects. Each node in the network (excluding the node that contains the local object) may have up to one stub per object in the original application. This way, any node can refer to any remote object in the system. Finally, the *proxy* object allows objects to migrate. Should an object be migratable (see Section 4.2.3), a proxy object will be allocated as well as the appropriate local or stub. The proxy holds a single reference to the local or stub object, and all references to that object pass through the proxy. This way if an object should migrate it is necessary only to update the reference within the proxy to refer to the new implementation. Since the majority of objects within an application never migrate, and we allocate proxies only for those that *may* migrate at some point, the majority of accesses do not incur this indirection.

Additionally, we extract the static parts of the original application from their rewritten counterparts. Static data is be unique within the system; individual nodes must not maintain their own, possibly inconsistent, versions of static state. To this end we create a *static singleton* object for each class that contains static data. These singletons are managed by the run-time library, and are guaranteed to be globally unique. We discuss this further in Section 4.3.3.

The second aspect of class rewriting involves copying and transforming the contents of the original class to the new local class. All object references must be re-typed to refer to rewritten interfaces rather than to the original classes. Additionally, field accesses are transformed to call get and set methods on the interface, rather than directly reading and writing fields. Finally, the method bodies are modified to call out to the run-time library to perform any additional functions required for distribution. These transformations are discussed in depth in Section 3.3.

When the RuggedJ class loader has rewritten a class, it presents only the transformed version for loading into the Java VM. The VM never sees the original class, which eliminates the possibility of conflicts between modified and unmodified classes.

1.2.4 Run-Time Infrastructure

The second component of a RuggedJ node is the run-time infrastructure. The run-time provides library functionality to rewritten bytecode, and is responsible for coordinating the activities of each node's rewritten classes to execute the application as a whole. The various functions of the run-time are performed per-node, minimizing the network traffic generated by the system itself.

The first task of the run-time library is to simplify rewritten bytecode. Some functionality is best hidden behind an abstract API, avoiding the need to generate complicated and implementation-specific bytecode sequences inlined in transformed application code. As an example, our messaging system relies upon Java's TCP sockets. Managing these sockets within rewritten bytecode would be a complex and error-prone task. Rather we present a simple interface to the run time library, and so can implement the socket interactions as Java source. Additionally, abstracting communication as library functionality made it simple for us to change the underlying networking infrastructure from an MPI library to Java sockets when we determined that communication was a bottleneck (see Section 4.1.2).

The run-time library is also responsible for tracking remote objects. As we will describe in Section 4.2.1, shared objects in RuggedJ are tracked by unique identifiers (*UIDs*). Each node must track those shared objects to which it has a reference, as well as the node upon which that object is located. This ensures that we do not create spurious stubs, and allows us to connect a stub to the correct remote implementation. Additionally, RuggedJ allows objects to migrate to maximize data locality. The run-time library coordinates migration, moving objects from node to node and updating references within proxies. It also coordinates with other nodes to ensure that all references go to the correct node. RuggedJ allows for replication of immutable state (discussed in Section 4.2.2). The run-time library fetches immutable content on demand when required by the local node, creates local replicas of such state, and ensures that each immutable object is represented by a single local replica.

Finally, each node's run-time library coordinates with corresponding libraries throughout the network to perform globally synchronized actions, to disseminate network state

information and to cache RuggedJ's metadata in order to minimize network traffic. For example, RuggedJ maintains static singletons for those classes with static data. These singletons are created as required on the first node that refers to them, spreading the singletons across the network to minimize bottlenecks and improve locality. The creation of a static singleton requires coordination between run-time libraries to ensure that the singleton is globally unique. Once the singleton is created, its location is propagated through the network using the tree-based communication hierarchy, and is recorded on each node. This reduces communication with the head node when each node first refers to the singleton.

1.2.5 Partitioning

In order to distribute an application across a RuggedJ network, we must determine which objects are to be allocated upon which nodes. We refer to this process as *partitioning* the application. RuggedJ provides a partitioning interface to which developers provide an application-specific policy. We believe that the application developer is in the best position to provide an optimal partitioning, guided by the output of our whole-program static analysis.

Application developers provide a partitioning plug-in class that extends RuggedJ's abstract `Partitioning` class. Each node has its own instance of the partitioning policy, allowing most decisions to be made locally and so reducing network communication. The partitioning policy has access to the local node's run-time library, allowing it to make decisions based on the dynamic state of the network. Additionally, partitioning policies have access to RuggedJ's communication infrastructure, allowing messages to be passed between instances of the policy.

We will discuss the interface to, and capabilities of, the partitioning system in Section 4.4, and cover some of the strategies that we employ when developing a partitioning policy in Section 5.2. We generally find that simple policies perform well: we identify the root of a distribution unit (typically the `Runnable` object that encapsulates the work of a single thread) and allocate instances of this class on remote nodes. By default, any

subsequent allocations are performed locally, so objects related to that distribution unit are automatically placed on the same node. We determine the nodes upon which to allocate distributable units by their capacity (the number of cores available to the node, determined by introspecting at run-time) and by the load level of the network (the ratio of the number of distributable units to the total capacity of the network). The majority of policies that we have developed for the benchmarks discussed in Section 5.3 have been expressed in a few dozen lines of Java.

1.2.6 Summary

Developing applications that perform well on RuggedJ requires little programmer effort beyond that needed to develop single-machine horizontally-scaling applications (as we discuss in Chapter 5). However, the RuggedJ framework offers significant benefit to application developers over manual distribution:

Standard Java. Applications that target RuggedJ are written in standard Java, with no additional libraries, language features or annotations. Developers do not need to learn new syntax, and applications are not tied to one specific infrastructure. Additionally, legacy code (such as scientific computation packages) that were not written with distribution in mind can be integrated into distributed applications without modification.

Deployment on arbitrary networks. RuggedJ applications are not tied to a specific network configuration. Our partitioning system allows developers to write dynamic partitioning strategies that adapt their application's partitioning to the current network.

Caching and migration. RuggedJ automatically caches immutable data and allows developers to specify object migration through the partitioning policy. Hand-coding such behavior would be tedious and error-prone, and is unlikely to produce the same level of performance.

Object tracking. Developers need not implement the mechanics of object location management such as looking up the location of objects, maintaining their sources and destinations during migration, and customizing code for local or remote objects.

Simple remote invocation. Remote method invocations in RuggedJ are transparent to the developer. Applications are not explicitly aware of the object locations, and so do not need to handle calls to remote objects. This is particularly useful in the presence of migration; developers need not determine at run-time whether an object is local or remote at a given point in the program.

Immutability control. Partitioning policies give developers the ability to label classes as functionally immutable. Instances of such classes may not be statically determinable as immutable (they may contain non-final fields that are modified outside the constructor), but the developer knows that they will never be modified after an initial set-up phase. This allows the developer to control replication more precisely.

Finally, applications developed for RuggedJ can make use of standard tool chains and development methodologies. Since RuggedJ operates only when the application is deployed, programmers can use any Java development and debugging tools.

1.3 Contributions

While the concept of transparent distribution is not in itself novel, our approach to the problem differs in several key ways from previous work (as we will discuss in Section 2). This dissertation presents several main contributions:

Novel infrastructure. We present a user-level infrastructure that transparently distributes large Java applications running on standard JVMs. This is the first transparent Java distribution system that dynamically adapts to arbitrary network configurations, using a programmer-defined partitioning strategy rather than a static partitioning.

Object model. Our object model allows virtualized access to the objects throughout a system, enabling us to interpose arbitrary code that tailors the functionality of the application executing on that system.

Whole-program transformation. We enumerate the major barriers to transforming applications, specifically the presence of native code and system library code that cannot be rewritten. We offer transformation templates that allow such classes to conform to our object model, and discuss of how transformed classes can interoperate with unmodified code. We have developed a classification algorithm that determines which source classes should be transformed in which ways.

Run-time mechanisms. We demonstrate how immutable data can safely be replicated in a distributed system, and how objects can be migrated and tracked transparently to the developer.

Distributed semantics. We maintain Java's original semantics in the face of distribution, including an elegant remote monitor implementation, a system to maintain global uniqueness for static data and an infrastructure for managing exceptions.

Partitioning interface. We describe the interface to our partitioning system that gives developers the flexibility to take full advantage of the RuggedJ infrastructure. Our partitioning framework is designed so that plug-ins can be extensively tailored to the individual application, while simple policies can be implemented in a few lines of code.

Application properties. We characterize the properties of *distributable* applications, indicating classes of applications that will perform well when distributed and the design features that can lead to poor performance. We discuss optimizations that improve distributed performance, both in the general case and when targeting RuggedJ in particular.

Performance study. We demonstrate scalability for realistic benchmark applications on a large-scale cluster of 48 cores over 3 machines.

2 BACKGROUND

RuggedJ draws inspiration from a number of prior systems that have explored Java distribution. In this chapter we will survey the major systems that we have learned from, and discuss some alternative strategies to distribution.

2.1 Transparent Distribution

Transparent distribution (distributing an application with little or no input from the original developer) is a promising approach to developing distributed applications. Several systems have implemented transparent distribution for Java in the past, developing some of the techniques that we have used in RuggedJ.

2.1.1 J-Orchestra

J-Orchestra [Tilevich and Smaragdakis, 2002, 2009] is a transparent Java distribution system that formed part of the inspiration for RuggedJ. Indeed, J-Orchestra influenced many of RuggedJ's early design decisions. However, J-Orchestra's fundamental goal is different from RuggedJ's: We distribute applications dynamically across arbitrary network configurations, while J-Orchestra aims for "resource-driven distribution," where one shares an application between a small, fixed set of machines with specific capabilities. For example, a transformed system may perform calculations on a back-end server, while displaying its user interface on a PDA. The design of each system reflects these differing objectives. J-Orchestra uses a static design to execute fixed client-server applications or for rapid prototyping [Liogkas et al., 2004]. The overall design of J-Orchestra fits well for the applications that they target, which generally consist of client/server communication or applications with a UI running on a separate host than the main processing work [Tile-

vich et al., 2005]. The primary goal is to take advantages of specific resources on different machines, rather than to distribute a large processing job across multiple back-ends.

The RuggedJ object model is partly derived from J-Orchestra, which relies upon a similar if somewhat more simple model. Classes within J-Orchestra are determined by the user to be *anchored* or *mobile* [Tilevich and Smaragdakis, 2002]; mobile classes are those that can be remotely referenced or can migrate, while anchored classes must remain fixed on a single machine. Mobile classes are replaced by rewritten *proxy* classes that allow for local or remote access, and any direct data accesses (getting and setting fields) are redirected through accessor methods. These proxies provide the same location-hiding function as our interfaces. A proxy instance can refer to either a local or to a remote object, requiring no special modifications when referring to such objects. The proxy class assumes the original name of the class, meaning that it cannot be elided when an object must be remotely-referenceable but does not migrate (an optimization that has proven effective in RuggedJ). J-Orchestra handles unmodifiable (system) code by wrapping. It creates proxy classes that encapsulate a reference to an associated system class which can then be referred to by rewritten code [Tilevich and Smaragdakis, 2006]. This approach is the same as our *Wrapping* system code template (discussed in Section 3.4); we found that wrapping and unwrapping incurred a performance overhead, and so developed additional techniques to handle system code.

J-Orchestra partitions applications at the class level; instances of anchored classes are always allocated on the same machine. This simplifies the object model, since they can predict ahead of time which references will be local and which will be remote (classes can be *co-anchored* to ensure that all their references are local). However this means that instances of anchored classes cannot exist on multiple machines, limiting the partitioning strategies available to developers. J-Orchestra's rewriting system is static and performed offline before the application is deployed. Each class is rewritten as Java source code according to its designation (anchored or mobile), compiled to bytecode and delivered in a per-site `jar` file to each machine in the cluster. This collection of classes represents the transformed version of the application appropriate to that node, which runs it as a regular

application. This strategy works well for J-Orchestra; it removes the need for load-time rewriting, and their fixed-network partitioning does not suffer for the lack of dynamism. In contrast, RuggedJ's dynamic partitioning system allows per-instance decisions, allowing us to allocate instances of a given class on arbitrary nodes within the network. Not only does this let us take advantage of current network conditions that cannot be predicted ahead of time, but it also allows us to perform load-balancing by distributing key objects of a given class across the network.

J-Orchestra uses a run-time infrastructure that performs similar functions to that of RuggedJ. A large part of the run-time's functionality is creating remote objects. J-Orchestra uses an RMI-based distributed object factory that runs on each node and reflectively creates instances of mobile objects. This differs from RuggedJ's object creation approach where the majority of instances are created locally using Java's standard **new** operation, with only those few objects that are explicitly determined to need remote allocation incurring the expense of reflective creation. The run-time also contains support for threading and synchronization. J-Orchestra uses RMI for remote execution, and so cannot implement the thread affinity-based synchronization approach that we discuss in Section 4.3. Instead they implement a separate run-time library that mimics the actions of monitors, extending RMI calls to include a global thread equivilance class identifier that determines which thread should acquire any monitors during a method's execution. They found this mechanism to incur an overhead of 5.5-12% [Tilevich and Smaragdakis, 2004].

The J-Orchestra run-time system does not manage exception handling, rather allowing the user to supply custom error recovery blocks within the proxy. This approach offers more flexibility than the automatic exception handling that we describe in Section 4.3 but lessens the transparency of the system. The run-time also does not intercept the majority of reflective calls. While the system designers argue that fully correct support of reflection is possible using a mechanism similar to that discussed in Section 4.3.2, they do not expand on the claim; it is unclear how they would handle such issues as reflective access to wrapped objects.

2.1.2 Terracotta

Terracotta [Terracotta Inc.] is an open-source JVM-level clustering system. It has several similarities to RuggedJ, requiring no specific API for developers to implement, and using bytecode-rewriting to allow mostly-transparent distribution. Terracotta also targets a similar class of applications to RuggedJ: where J-Orchestra was optimized for applications that distribute across small, fixed networks, Terracotta targets large distributable applications that can run on large clusters. However, the Terracotta approach differs from that of RuggedJ in several key aspects.

Users define Terracotta's shared object graph as a closure of objects reachable from distribution *roots*. All objects that can be reached from these roots are considered to be shared. By contrast, RuggedJ considers all objects to be potentially remotely-referencable, and so does not distinguish between shared and non-shared objects. Terracotta root objects have different semantics from regular Java objects: the value of a root field may not be changed, as doing so would affect the shared object graph. Objects reachable from a root are referred to as *clustered*. Such objects have a cluster-wide identifier and so can be remotely referenced. Terracotta also uses this shared graph for persistence; clustered objects can be persisted without additional specification.

The Terracotta run-time system uses a client-server approach that differs strongly from that of J-Orchestra or RuggedJ. A central server (which can be distributed to reduce the bottleneck that it presents) manages all clustered objects (that are located on this central server) as well as global activities such as locking. Terracotta uses a transaction mechanism to perform work on the client systems. A remote thread locks a clustered object on the server, starting a transaction. Any necessary data is replicated on the client node, which performs local work on the safely-locked object. Once the transaction is complete, the client releases the lock and updates the cannonical object on the server, which propogates the changes to any remote replicas. This transactional, client/server approach differs from RuggedJ's peer-to-peer system, where cannonical versions of objects are spread throughout the system.

2.1.3 Addistant

Addistant [Tatsubori et al., 2001] enables the distribution of "legacy" Java applications (the developers define legacy as any Java software written without distribution). The system makes use of load-time bytecode rewriting using the Javassist transformation tool, and provides a run-time system. It requires no modification to the Java VM. Developers specify the locations of objects at the class level in a separate policy file; the authors claim that it is not realistic to individually specify where each object is located. The Addistant run-time system has the interesting feature that it automatically delivers rewritten source code to the respective nodes, simplifying application deployment.

The major contribution of the Addistant system is its object model. Like RuggedJ, Addistant uses proxies to forward remote references to the appropriate objects. They develop a classification that allows system code to integrate into distributed applications, based on two properties. *Modifiability* refers to the capacity of their tool to rewrite bytecode; we discuss a similar concept in the differentiation between user and system code. *Heterogeneity* refers to the references that a class holds; a heterogeneous class can refer to both local and remote objects. Based on these two criteria, they define four approaches to developing proxies. The *Replace* approach is usable when a class is modifiable and non-hetrogeneous. It assigns the class to one node and generates a proxy with the same name on all remote nodes. The *Rename* approach is used when the class is unmodifiable, but is referred to only by modifiable classes. In this case the system creates a proxy with a different name, and rewrites all references to point to this proxy. The *Subclass* approach allows heterogeneity: the proxy is a subclass of the base class. References pointing to the base class can instead refer to the proxy. Finally the *Copy* approach is used for primitive and immutable objects, with replicas passed around the network.

Addistant takes the same approach to object equality as RuggedJ; equality is guaranteed by ensuring that exactly one proxy object per host refers to any given master object. It also uses a similar thread affinity system to RuggedJ, ensuring that callbacks from remote methods are handled by the same thread.

2.1.4 AIDE

The AIDE system [Messer et al., 2002] proposes "offloading" of work from low-power computational devices such as PDAs. Monolithic applications are transparently distributed to make use of available remote resources. AIDE is implemented in Java using a modified version of HP's Chai virtual machine. It uses a class-level partitioning system where all instances of a given class are colocated. The partitioning is determined at run-time through VM instrumentation; a weighted execution graph is built up from instrumentation data. A graph partitioning is computed at periodic or resource-based trigger points. Lightly-connected components are considered candidates for offloading, while strongly-connected components have frequent interactions and so should be colocated.

AIDE bypasses some of the rewriting issues that RuggedJ faces by increasing the function of the head node. All native methods are executed on the original JVM, and while static functions can be executed anywhere, static data remains on the original VM. This simplifies the implementation of static data, but could cause a bottleneck at the original, presumably low-power, VM. This overhead is difficult to determine; while an implementation of AIDE was built, the authors report numbers from an emulator.

2.2 Java Distributed Shared Memory

Distributed Shared Memory (DSM) systems use a cluster of modified Java VMs to implement a single shared-memory image. While this does not directly compare to RuggedJ's transparent distribution approach (as it involves VM modification), such systems can run similar applications in a distributed manner.

A number of Java DSM systems use the Homebased Lazy Release Consistency model (HLRC) [Iftode, 1998, Samanta et al., 1998]. HLRC is a page-based virtual memory consistency model that assigns a "home" node to each shared page. It computes difference maps (*diffs*) for each dirty page at the end of a specified interval (such as during a release operation) which are immediately applied to all other versions of the page. This model al-

lows Java VMs to cache local updates while ensuring that memory is consistent at defined points.

Java/DSM [Yu and Cox, 1997] was the first distributed shared memory implementation for Java. It was built on top of the Treadmarks system [Keleher et al., 1994], a DSM implementation that uses a a homeless LRC protocol to provide a shared memory abstraction for Unix applications. The Java/DSM heap is allocated in the Treadmarks shared memory area. Data is shared per-page, and types are modified to account for hardware differences such as endianness. MultiJav [Chen and Allan, 1998] is modified Java VM that implements a Java DSM. MultiJav differs from Java/DSM in the unit of sharing: sharing in MultiJav is per-object, with synchronization performed by release-consistency. Shareable objects are detected by the VM, with no programmer annotations necessary, and are referenced through handles mapped to machine memories.

cJVM [Aridor et al., 1999, 2000] is a cluster-aware Java Virtual Machine implementation that uses an optimized object model to improve performance. A master object is located at the node where an object was created. All other nodes in the cluster have proxies that refer to the master. A proxy can have multiple implementations, with the most efficient chosen at class-load time. The cJVM object model supports object and thread migration, caching through replication and remote method invocation.

cJava [da Silva et al., 2003] is a Java DSM based on HLRC in which each node executes an instance of a modified VM. It runs unmodified multi-threaded programs in a distributed manner with no additional programmer input. Each JVM contains a Distributed Object Manager that allocates objects and manages the global thread space, an Event Manager that controls communication between JVMs and a Thread Manager that oversees synchronization and thread creation. When running on multiple VMs they observed sub-linear speedup, which they attributed to a mismatch between the HLRC model and Java's language semantics. The Cooperative Java Virtual Machine (CoJVM) [Lobosco, 2003, Lobosco et al., 2005] is a similar Java DSM that uses the HLRC model. It uses selective diffing to update only those pages that contain dirty Java objects, and lazy home allocation to minimize bottlenecks.

2.3 Language-Based Distribution

RuggedJ distributes Java applications, maintaining the semantics of the original language in a distributed context. This approach has its advantages: developers are largely familiar with Java, and efficient implementations of the Java Virtual Machine are widely available. However a different approach would be to design a new language from the ground up with distribution in mind. This would allow for extra flexibility in language concepts than was available to us, and could greatly simplify implementation by imposing distribution concepts into the language, compiler and run-time systems.

The Emerald programming language [Black et al., 1986, 2007] implements many of the same concepts as Java. It is an object-oriented language where data is encapsulated as object state, including primitive data such as integers or booleans. Classes implement abstract types, similar to Java's interfaces, and objects may have an optional process attribute that allows them to execute code in a concurrent thread. Unlike Java, however, Emerald contains direct support for distribution. Instances have an explicit location attribute and unique name (similar to RuggedJ's unique object identifiers) that allow them to be tracked across the network. Objects can be declared immutable by the developer, allowing them to be replicated across multiple nodes, and objects can be migrated from node to node.

Emerald's object model presents an advance over earlier distributed languages such as Argus [Liskov et al., 1987] or Eden [Black, 1985] in that it uses a single representation for local and remote objects. Previous systems had required that developers implement two versions of a given object if it was to be used in a local and remote context. In Emerald developers supply a single object specification that is transformed by the compiler into one of three representations: *Global* objects are reachable from remote nodes, *Local* objects are colocated with another object, and *Direct* objects are inlined into an enclosing object. Each implementation inherits from a common abstract type, allowing developers to refer to them in a uniform manner.

While the Emerald object model makes locations transparent during invocations, developers can explicity obtain object location information. Each object implements several

operations: a developer can *Locate* an object (tracked by a run-time system that uses forwarding references to resolve migrations, falling back to a broadcast system if the information is unavailable [Jul et al., 1988]), *Fix* or *Unfix* an object (anchoring it to a particular node), and *Move* an object from one node to another. Parameters to remote method invocations are generally passed by reference, with developers able to specify arguments to be migrated to the invoking node.

X10 is a modern object-oriented distributed programming language that was designed to enable large, scalable applications to run across clusters of high-end computers [Charles et al., 2005]. X10 introduces the concept of *places*: an execution environment with a finite number of threads and a bounded region of shared memory that is accessible with uniform time to the local threads. Accesses to remote places are performed using *futures*. An asynchronous request is spawned by the local thread and returns immediately, with the result of the access supplied later. This allows X10 to hide some of the latency of remote data accesses. Computation is performed using *activities* which can execute synchronously or as part of a future.

Synchronization in X10 is implemented using *clocks*. A clock represents a global barrier with which activities can register. Activities reaching the end of a specific clock phase must wait until all registered acivities have *quiesced* by indicating that they are ready to proceed. X10 clocks generalize the concept of barriers by allowing a given activity to synchronize on multiple clocks, while still guaranteeing such programs to be free of deadlocks [Agarwal et al., 2007].

Fortress [Steele, 2006] is a high performance language designed to offer the same portability properties as Java. Fortress is designed to be highly parallel, with language constructs that make parallelism the natural mode of development. Multiple operands to an operator, or expressions within a tuple, can be executed in parallel using any resources available. Fortress uses a work-stealing technique first developed for Cilk [Blumofe et al., 1995] to distribute work between idle threads; expressions to be evaluated are placed on a work queue from which other threads may scavenge work when idle. Fortress was designed to

eventually execute across clusters of nodes, but that direction is not currently being pursued.

2.4 Other Java Distribution Systems

There have been a number of different approaches taken to distributing Java applications, with varying degrees of transparency to the developer. We provide here a summary of the major systems.

The standard mechanism for distributed computation within Java is Remote Method Invocation (RMI) [Sun Microsystems, Inc., a]. Java RMI provides an API that allows programmers to create and manipulate remote objects directly. Classes that may be used remotely implement *remote interfaces* that define the operations that can be performed by remote clients. Remote objects are represented by stubs that forward any accesses to the original object. Java RMI makes distribution fully explicit; developers must track the locations of objects and must be aware of object location when designing their applications.

JavaParty [Philippsen and Haumacher, 2000, Philippsen and Zenger, 1997] is a a source-level transformation system that adds support for remote objects to Java. Classes in JavaParty can be declared as `remote` (using a newly-defined modifier) and so are visible and accessable from any other node in the network. The run-time system takes care of placement and communication, and removes the need to register shared objects; a remote object is globally visible upon creation. JavaParty is implemented on top of RMI as a pre-processing step to a Java compiler. It transforms the `remote` keyword into RMI stubs, providing a simpler interface to Java distribution. JavaParty introduces several features that were implemented in J-Orchestra, and later in RuggedJ. It generates interfaces for shared classes with different implementations for local and remote version. It also separates the static parts of classes into a new generated class that can be managed by the run-time system. JavaParty supports object migration for spatial locality.

Do! [Launay and Pazat, 1998a,b] transforms annotated parallel Java programs into distributed Java programs. It uses a preprocessor to create new clases, and a run-time system

that manages actual distribution, remote object creation and so forth. Original programs are developed in terms of `TASK` objects that operate over `COLLECTION` objects that represent the application's data. This explicit Single Intruction, Multiple Data (SIMD) model ensures that applications are distributable and provides ready distribution points.

Doorastha [Dahm, 2000a,b] is an extension of Java RMI that allows fine-grained optimization while preserving standard Java semantics. It allows a per-object determination of argument passing semantics, whether an object should be passed by reference or by copy. A given object can be passed using different semantics at different times, with passing by reference the default. The system also supports object migration as an additional optimization, allowing objects that are not explicitly passed as arguments to move between nodes. Annotations allow the developer to specify how much of an object's transitive closure is to be copied. Doorastha is implemented using specially-formatted comments as annotations that are read by a custom compiler. The system also includes a custom run-time layer that exists on top of RMI.

Java// (pronounced "Java Parallel") [Caromel and Vayssière, 1998, Caromel et al., 1998] is a set of library classes that allow code to be executed on single-processor, multiprocessor, or cluster machines. Java applications that are developed using these libraries intantiate the appropriate version at run-time. Objects that are set as *active* through method calls have proxies created on remote machines, and method calls automatically redirected. The Java// system supports transparent futures, allowing multiple outstanding calls that hide network latencies. Java// uses an abstract concept of a *node* to refer to partitioning units; multiple nodes can represent machines in a cluster or can be colocated on a single multiprocessor. This way partitioning choices are built into the application whether it runs in a distributed manner or not.

Javanaise [Hagimont and Louvegnies, 1998] is a library-based Java distribution system that does not perform bytecode rewriting or rely on a modified Java VM. The developer defines clusters of related classes, which are colocated to minimize communication. Each application class has a set of proxies that serve as an entry point to the Javanaise run-time system: `proxy_in` (located on the same machine as the object) and `proxy_out` (located on

remote machines). Proxies are provided by the developer, and implement Javanaise marker interfaces. The run-time system provides synchronization and coherence between threads, and communication between proxies.

Object Request Brokers (ORBs) are middleware systems that allow programs to call one another across a network. The ORB provides a standard set of call semantics that allow systems running on different platforms and coded using different languages to interoperate. The Common ORB Architecture (CORBA) [OMG, 1992] defines a standard interface and set of features that are implemented by many ORBs.

2.5 Work Related to Key RuggedJ Features

There are several projects with goals other than distribution that use similar techniques to those in RuggedJ. In this section we discuss such related work, focusing on the similarities and differences from our implementation in RuggedJ.

2.5.1 Object Model

The *Infer Type* refactoring [Steimann, 2007] aims to increase the reusability of code by typing fields as interfaces rather than as classes. The refactoring takes an object reference and generalizes it to the minimal interface that encapsulates its protocol. By retyping field references using this new interface, which is then implemented by the original referred class, Infer Type allows references to be specified by the minimum set of operations required, reducing the effort needed to reimplement this functionality at a later time. A key feature of the Infer Type refactoring is producing the minimal set of functionality required of a reference; while RuggedJ generates interfaces for each class in the original application, it does not modify their protocols.

Java introduced dynamic proxies as part of the standard language specification's reflective API version 1.3 [Sun Microsystems, Inc., b]. Such proxies allow developers to interpose arbitrary code around method invocations by redirecting accesses through proxy objects. Java's dynamic proxy mechanism works only for fields typed as interfaces, severely

limiting the usefulness of such techniques. Uniform dynamic proxies [Eugster, 2006] addresses this limitation by extending support to allow proxies for class-typed objects. This is implemented using a series of bytecode transformations that produce a unified object model, redirecting field accesses through accessor methods.

The Automatic Test Factoring system [Saff et al., 2005] produces "mock" versions of objects which return memoized results from a previous measuring run, allowing developers to speed up the testing of individual application components. Their system uses the same interface technique that allows us to refer to proxy and local stubs transparently; in their case the interfaces allow them to switch real classes with their mock equivalents, determining which parts of an application are to be tested. The Test Factoring system differs in the way it handles system code. Rather than redirecting through wrappers or extending classes, they directly rewrite the system library to include mock objects. This is not feasible in our system, due to the limitations of visibility between class loaders. Such rewrites are possible only if classes are not renamed, and any referenced libraries are stored in the boot class path.

2.5.2 Whole-Program Transformation

The issue of rewriting system code has been considered in the past. The Twin Class Hierarchy (TCH) approach [Factor et al., 2004] copies relevant system classes into a user-level package, which can then be rewritten and referred to by rewritten user code. Because the original system classes remain unchanged, any instrumentation inserted into the rewritten versions can safely refer to system classes without affecting the statistics gathered or causing an infinite loop. The TCH system does not allow rewritten system classes to interact with the original classes, making it too limited for our needs. Additionally, the TCH approach requires custom wrappers for all native methods. This approach does not scale, and could require that separate wrappers be written for different implementations of the standard class libraries, compromising ease of deployment over heterogeneous Java VMs.

2.5.3 Application Partitioning

Pangaea [Spiegel, 2000, 2002, 1999] acts as an automatic partitioning front-end to several distribution systems (including Java RMI, JavaParty, CORBA and Doorastha). It functions over distributable applications designed as a single-machine concurrent program, and performs a guided static analysis to determine distribution policies. The developer provides a starting point to the analysis; boundary objects are assigned to particular machines, and used as the base for determining distributable units. The static analysis optimizes the partitioning using whole-program knowledge, such as immutability, dynamic scope and phase behavior leading to object migration [Busch, 2001]. Pangaea could conceivably be used as a front-end to RuggedJ, generating a partitioning policy based on its static analysis. However RuggedJ's partitioning interface allows for significantly more flexible policies than would be generated by Pangaea. For example, we can take advantage of programmer knowledge to mark statically-mutable classes as functionally immutable, and we allow the developer to insert custom migration triggers to maximize locality. Thus, a Pangaea-generated partitioning policy would not take full advantage of RuggedJ's capabilities.

3 CLASS TRANSFORMATION

Class transformation is key to our distribution system. Injecting distribution logic into regular Java code allows classes to interoperate with remote objects and with the RuggedJ library without modifying the underlying Java virtual machine. We perform extensive transformations on each of the classes that make up the original application: we generate an interface that encapsulates the protocol of the class and three implementations of this interface to represent local, remote and migratable objects. Additionally, for classes with static data we create a static singleton that represents this content, generating a further interface and three classes. Finally, we rewrite the contents of the original classes to be aware of these new classes and to work within a distributed environment.

We define an additional two goals in the transformation process. First, we aim to keep the rewritten bytecode as simple as possible. This stems from the practical difficulties inherent to debugging bytecode; the simpler the rewritten bytecode the more straightforward the debugging process. Additionally, overly-verbose bytecode transformation sequences are more likely to lead to complex interactions where generated bytecode sequences are accidentally modified by subsequent transformations. The second goal is to optimize transformed code for local execution. This is a result of two constraints: the vast majority of object accesses in the distributed system should be to local objects, and the overhead of remote invocations is such that optimizing bytecode will do little to affect the overall performance penalty in these cases.

We perform class transformation at the bytecode level, using a custom Java class loader. Bytecode transformation offers several advantages over source-level modification. We transform our modified classes on-demand, without consideration of inter-class dependencies. Modified Java source code would have to be compiled, which would require that the whole program was rewritten ahead of time; we take advantage of incremental transformation to optimize classes for their location in the network. Additionally, bytecode is a

significantly less complex representation of an application, since Java constructs and variables are collapsed to stack and register operations. This makes the transformation process simpler, as there are fewer cases to handle.

There exist several strategies to rewrite bytecode. Aspect oriented programming (AOP) is a design methodology that aims to separate cross-cutting concerns from the main logic of an application [Kiczales et al., 1997]. An *aspect* is a transformation that is inserted at specific, well-defined points in an original application (known as *pointcuts*), augmenting or replacing the existing code. This way, aspects can be used to implement features such as logging or error handling separately from the main application. The most commonly-used implementation of AOP is AspectJ [Kiczales et al., 2001a,b] which includes both a source-level and bytecode-level aspect weaver that rewrites original classes. AOP suffers from a lack of low-level control; aspects are specified in terms of the classes that they modify, and allow *advice* to insert or modify code that corresponds to specific pointcuts. This matching process makes it difficult to design general aspects that perform specialized context-specific rewrites on arbitrary classes. MetaAspectJ [Huang and Smaragdakis, 2006] aims to remedy this issue by providing an aspect-generating framework that can create specific aspects programatically. However, even with this additional tool, AOP is capable only of modifying existing classes; it cannot be used to generate the new classes required by a system such as RuggedJ.

A similar but more flexible, approach is that of Javassist [Chiba, 2000, Chiba and Nishizawa, 2003], where on specifies code transformations in Java syntax, which is then compiled with a custom compiler. This offers a lower-level interface to rewriting than AOP. However, we found that its on-demand compilation approach made whole-program modification difficult. Jinline [Tanter et al., 2002] is a related project that allows load-time rewriting of bytecode. It provides a version of AOP at the bytecode level, inlining a specified method body at a given bytecode location. Jinline provides static and run-time information to user-defined listeners, which are called whenever a matching bytecode sequence is encountered. JMangler [Austermann et al., Kniesel et al., 2001] intercepts and rewrites bytecode at load-time. It is able to work with user-level class-loaders by providing

a modified version of the `ClassLoader` class. JMangler is currently limited to Java 1.4, making it unsuitable for our needs. Barat [Bokowski and Spiegel, 1998] loads either bytecode or Java source and builds a complete AST. It performs name and type analysis on the code, making the results available for use in other rewriting systems. While the analyses provided by Barat would have been useful in developing RuggedJ, the system is currently limited to analyzing Java 1.1 class files.

Ultimately, we determined that ASM [Bruneton et al., 2002] supports a good balance of direct access to method bytecode while hiding awkward details such as management of constant pools and the selection of instructions with hard-coded local variable slots. These two abstractions vastly simplified the design of transformations and generated bytecode, making ASM more useful to us than the similarly-featured BCEL [Dahm, 2001]. Additionally, ASM supports the class file extensions specified in Java 6, allowing us to make use of the latest language features. As a result, we performed the vast majority of our transformations using ASM, with some minor additional modifications performed by a custom C library (see Section 3.4.2).

When transforming an application, we must account for Java system code. A user-level class loader cannot rewrite classes from the Java standard libraries, and so we cannot transform them in the same way as we would any other code. In accordance with our goal of simple bytecode transformation, we use a combination of four different techniques to handle library classes, removing the need for special treatment in rewritten bytecode:

Direct classes. We classify immutable and purely local objects as *Direct*, and refer to them without modification. Immutable direct objects can be replicated on each node in the network, and so are never remotely referenced. This means that they need not implement the RuggedJ object model.

Promotable classes. System classes that are not referred to by other system classes within an application are referred to as *Promotable*. Since all references to these classes exist in rewritable code, we can create a copy of the system class that we are free to rewrite.

Extending classes. For many system classes we can generate an *Extending* set of classes that implement the RuggedJ object model. This allows the object to be referenced by remote nodes in the same way as any other transformed class.

Wrapping classes. Finally, we can generate *Wrapping* classes that implement the object model without extending the original system class. This technique allows any system class to be remotely referenced, at the cost of wrapping and unwrapping overheads.

Classes in RuggedJ are transformed on demand, with a rewriting class loader on each node. This way we can transform classes differently on different nodes; if we know in advance that a class will only ever be allocated upon a single node we can rewrite all accesses from that node as purely local, and all accesses from any other node as purely remote. In addition to the rewriting class loader, we also use a Java Virtual Machine Tool Interface (JVMTI) agent to perform limited modifications to Java system code.

The remainder of this chapter is structured as follows: Section 3.1 outlines some of the terminology that we use when discussing bytecode transformation. Section 3.2 outlines the RuggedJ object model, while Section 3.3 discusses the bytecode rewrites that we use to support the model. Section 3.4 discusses how we integrate system classes into the RuggedJ object model, and Section 3.5 describes the implications these classes have on user code. Finally, Section 3.6 gives some quantitative evaluation of our system code support.

3.1 Terminology

We use (and extend) various terms to characterize Java classes, which we now briefly define.

3.1.1 System and User Classes

Figure 3.1 gives a simplified overview of class loading within our system. We split classes into two sets, *system* and *user* classes, depending on the class loader that defines them. System classes are those in the Java standard libraries, and so are loaded by the

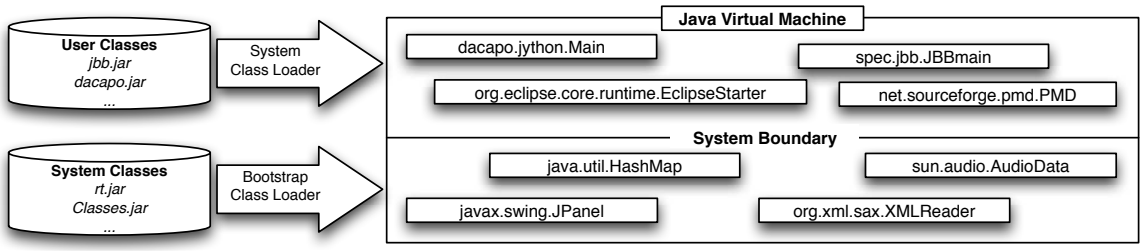


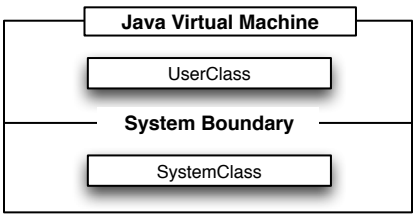
Figure 3.1.: User and system classes

virtual machine's *bootstrap* class loader [Liang and Bracha, 1998]. User classes, produced by the application developer, form the remainder of the application and are loaded by the user-defined *system* class loader. This distinction is vital when considering load-time transformation, as a user-level class loader can modify only user classes. We discuss Java's class loading mechanism, and its implications for our system, in Section 3.4.1.

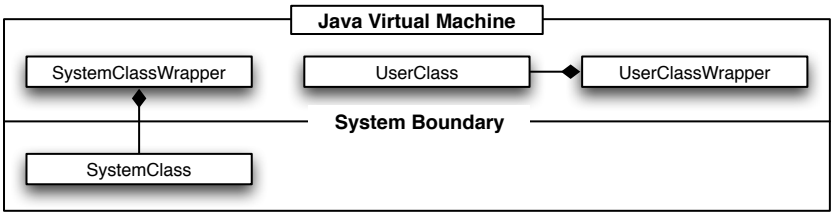
Within the Java VM itself we define the *system boundary* as a logical distinction between the two sets of classes; user classes exist above the system boundary, while system classes exist below. This abstraction is convenient when considering interaction between rewritten user and non-rewritten system code. We can enumerate the ways in which references can cross the boundary, and so ensure that rewritten references are never passed to system code.

3.1.2 Transformation

Figure 3.2 shows the implementation of *wrapping*; one of our approaches to handling system classes within a transformed application. In Figure 3.2(a) we see one system and one user class before applying any transformations. Figure 3.2(b) shows the result of wrapping each object. Class `SystemClassWrapper` contains a reference to the unmodified `SystemClass`. Since the wrapper was not generated by the bootstrap class loader it exists above the system boundary, with the reference crossing the boundary. In both cases, we refer to the original classes `SystemClass` and `UserClass` as the *base* class, while the two generated classes are *wrappers*.



(a) Before transformation



(b) After transformation

Figure 3.2.: Transforming classes

Additionally, within our system we refer to `SystemClassWrapper` and `UserClassWrapper` as *new* types. They are generated at load-time by our rewriting class loader, and thus can implement our object model. In contrast, `SystemClass` and `UserClass` are *old* types, as they come from the original application. Both sets of types are necessary; new types implement the uniform object model that allows all classes to be referenced in the same manner, while old types can be passed safely to system or native code that has not been rewritten to be aware of the presence of generated code. We maintain a strict separation of the two sets of types. User code refers exclusively to new types, while system code refers exclusively to old.

3.2 The RuggedJ Object Model

The ability to distribute an application in RuggedJ stems from the uniform object model that we apply to all objects. Figure 3.3 shows the transformation of a single user class `X` to conform to the RuggedJ object model. We discuss here the instance parts of the transformed class, and defer the static parts to Section 3.2.5.

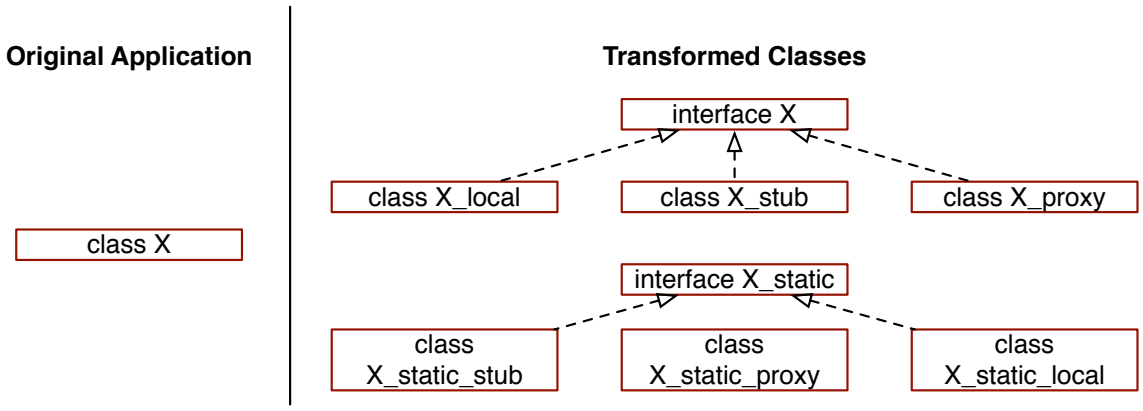


Figure 3.3.: The RuggedJ object model

3.2.1 Generated Classes

For each class within the original application we generate three classes and one interface. The generated interface, `X`, encapsulates the protocol of the original class `X`. It contains the signatures of all the original instance methods, along with new accessor methods for all the original instance fields. It uses the same name as the original class—this simplifies later rewriting of classes that refer to the original class `X`, since we do not need to update type names in method signatures, field definitions, or casts. Interface `X` is implemented by three concrete representations of the original class. The first, `X_local`, contains rewritten implementations of the instance methods of the original class, plus implementations of the new field accessor methods. In the rewritten application, an instance of `X_local` corresponds to an instance of class `X` from the original application: an `X_local` object holds all the data present in an old instance of `X`.

The second implementing class is used to refer to remote instances on other nodes: `X_stub` contains remotely forwarding implementations of all the methods of the new interface `X`, which simply call the corresponding method on a remote `X_local` instance. Within a distributed application, the local and stub instances have a $1 : n$ relation: any local object can be remotely referred to and invoked by stubs from the n nodes in the cluster.

The third (and final) new class is `X_proxy`. A proxy encapsulates a reference to either a local or stub instance, and its methods simply forward all calls to the target local/stub. Proxy indirection simplifies dynamic migration of instances to different nodes: a migratable instance is referred to by proxy, so upon migration only the reference in the proxy need be updated. Rewritten application code types all references to the three implementing classes using interface `X`. However we can bypass the proxy instance for objects that are known not to migrate. As all three classes implement interface `X` we can use them interchangeably without modification to any calling code. In RuggedJ we use programmer input to determine how to partition an application across the network.

All of the classes in an application can be adapted to implement the RuggedJ object model. As we shall see, we use several techniques to generate local classes. However each implementation strategy produces a class that implements the corresponding interface, allowing proxy and stub classes to interact with any style of local class in the same manner. As the designs of stubs and proxies do not vary between implementation techniques, they are so straightforward as to be uninteresting. We therefore focus our attention on the local classes.

3.2.2 Referring to Transformed Objects

Within rewritten code, we exclusively refer to values with generated interfaces using that interface. This allows us to vary the implementations of these interfaces among several alternatives (local, proxy, and stub classes) without impacting code elsewhere in the system.

Additionally, we use interfaces as a means of maintaining the class hierarchy from the original application. While some of the transformations we present in Section 3.4.3 do not maintain the original relationship between their local classes, we ensure that their generated interfaces do. Thus, since we refer to such classes exclusively by interface, we can perform subtype and instance checks correctly.

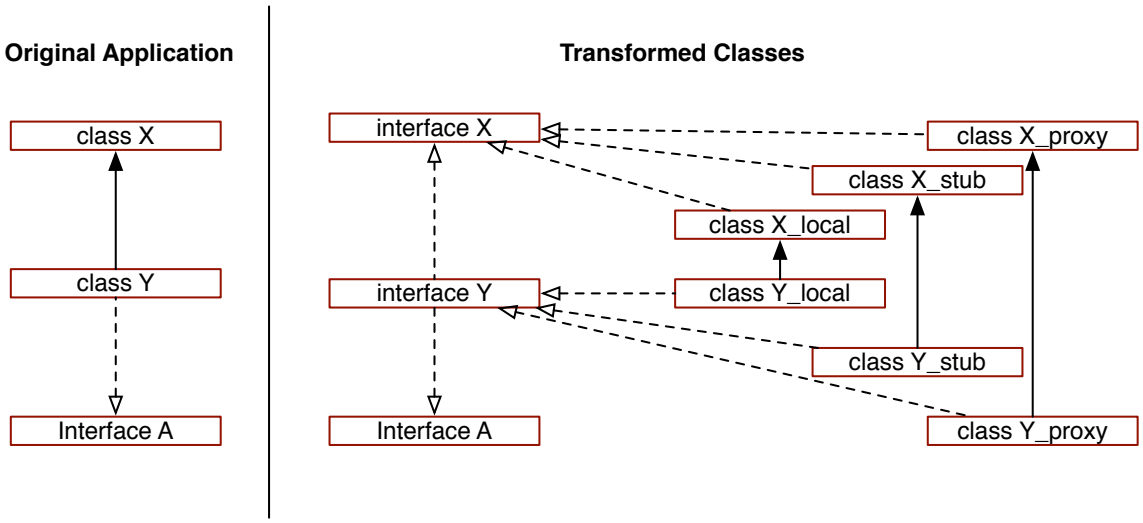


Figure 3.4.: Inheritance between transformed classes

3.2.3 Inheritance

As well as providing a mechanism by which we can reference different versions of a class uniformly, RuggedJ's generated interfaces maintain the inheritance relationships between original classes. Figure 3.4 shows the relationship between transformed classes (omitting static parts).

The original application's inheritance relationship between subclass `Y` of class `X` appears as the transformed interface `Y` extending interface `X`. Since rewritten code refers to objects exclusively by interface, this allows one to use any object that implements `Y` when the original code required an instance of `X`. Similarly, `CheckCast` or `InstanceOf` operations operate over interfaces, and produce the same results in transformed code as in the original application.

Each transformed class `Y_local`, `Y_stub` and `Y_proxy` extends the equivalent part of class `X`. This is not necessary to preserve the inheritance relationships of the original application. Other than when allocating instances, rewritten code never refers to these individual classes. Rather, this subclassing works to simplify the implementation of these classes. Without it, each class would have to contain the fields and implementations for ev-

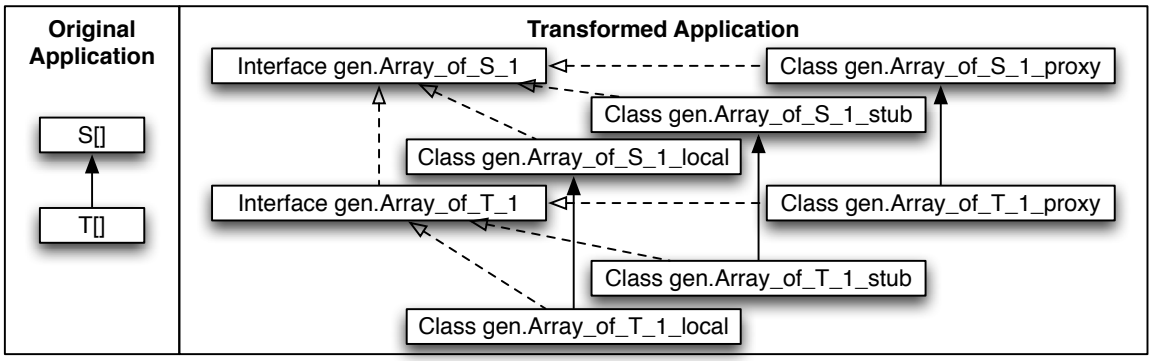


Figure 3.5.: Generated array types

ery method of the superclasses of its unmodified version, which would lead to duplication of code and overly-complex classes.

We do not transform interfaces from the original application (in general - see Section 3.4.3 for some exceptions) as they have no state that may be remotely accessed. However we must capture the relationship between a class that implements an interface; we do this by extending the original interface in the generated interface. This maintains the inheritance structure through generated interfaces in the same way that we do for class inheritance.

3.2.4 Arrays

We convert array types to new array classes, which allow us to refer to them as we do any other transformed class. The new array classes conform to the RuggedJ object model; we generate an interface, local class, stub class, and proxy for each, as shown in Figure 3.5. A one-dimensional array type `T[]` is represented by an interface `Array_of_T_1`, while a two-dimensional array type `T[][]` is represented by `Array_of_T_2`. An array type comprises both an element type and the number of dimensions of the array, so we encode both of these properties in the name of the new array types. Java defines subtyping among array types having the same dimensions only if the element types are subtypes. We capture

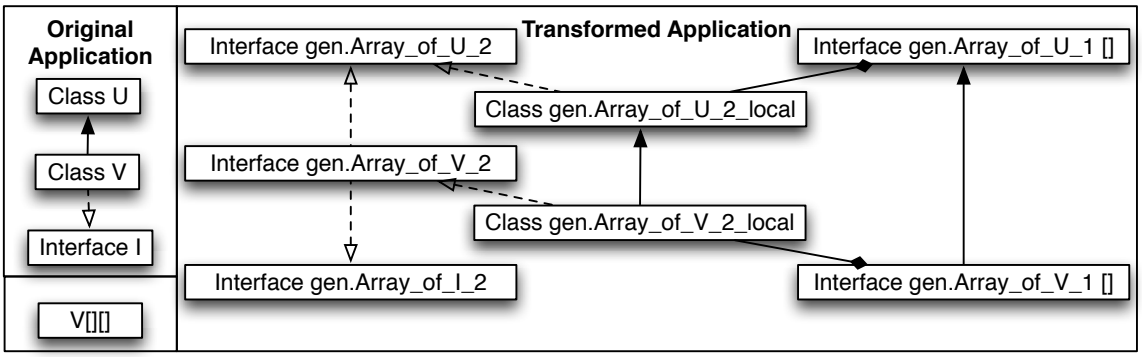


Figure 3.6.: Multi-dimensional arrays with interfaces

this by making any generated array class for a subtype directly extend the generated array class for its supertype (both having the same dimensions).

We implement arrays using wrapping: the generated array class wraps a regular Java array having the same component type as the wrapping array class. The implementation also provides methods to obtain the array `length` and to perform the standard operations that arrays inherit from `Object`, such as `clone`.

Figure 3.6 expands on the handling of arrays, showing the classes generated for a two-dimensional array type `V[][]` whose element type `V` extends `U` and also implements an interface `I`. We omit the new stub and proxy classes for clarity. This example highlights some interesting features of our generated classes.

Looking at the wrapped array within the local class, we see that the component type of the wrapped array is the same as that of the wrapper, with one less dimension. This mirrors the Java definition of arrays as a single dimension of components, where each component can be a sub-array. A useful consequence of this approach is that we do not place restrictions on the implementations of the components of the wrapped array, so long as they implement the appropriate interface. Thus, in RuggedJ, sub-arrays can be distributed across different nodes, regardless of the location of their enclosing array.

Figure 3.6 also illustrates that the old subtyping relationships between array elements and interfaces must also be represented in the new types. When passing array instances as

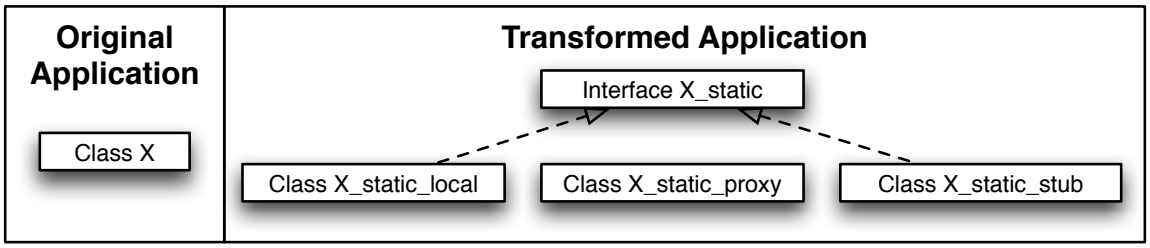


Figure 3.7.: Handling static data for distribution

arguments it is necessary for `Array_of_V_2` to implement `Array_of_I_2`. If an original method signature expects an array argument whose elements implement a given interface `I`, then in the rewritten new method we will expect an argument that implements some interface `Array_of_I_`n (for some dimension n), so capturing the proper type constraint. Within that new method all `AALoad` operations are rewritten as `get` invocations on the argument. The type constraint ensures that any argument passed to the new method will have an appropriate `get` method to return a value implementing `I`.

3.2.5 Static Data

A class's static state presents a complication in a distributed setting, since an application must see just one version of the static state. Simply rewriting class fields as static fields in the transformed application will result in each node having a separate loaded class with that field, whose states will not be coherent across the nodes. We approach this issue through the use of *static singletons*. We extract the static parts of each class to form a single instance, which we handle as any other object within the system. The instance state of this singleton object represents the static state of the original class, and can be accessed from any node.

Since static singletons are required only to maintain a canonical version of static data, we do not need to create a singleton for a class that has no static fields. Our analysis shows that static singletons are required in only 18% of classes in the applications we studied.

Static singletons implement the RuggedJ object model as shown in Figure 3.7. Interface `X_static` complements the instance interface `X`; it contains the static members of original

class `X`. We transform the static members of the original class into instance members of `X_static_local`, and use the RuggedJ run-time library to ensure that only one instance of that class is ever created. Thus, simply rewriting all static invocations to use the static singleton ensures that the static data is indeed unique.

The stub class `X_static_stub` performs the same remote access function as its instance counterpart. The final class in Figure 3.7, `X_static_proxy`, acts as a per-node cache for the appropriate static local/stub object, and is never instantiated. Accesses to static data in the original application (such as via the `InvokeStatic` bytecode) are handled by the virtual machine, resolving the class name to access the appropriate data. In our rewritten version, however, we need a static singleton object upon which to invoke methods. Obtaining this reference through the library would be an expensive operation, requiring a hash table lookup for every static access. Instead we store the reference as a static field in the `X_static_proxy` class, which can be obtained through a regular static field access.

3.2.6 Hand-Coded Classes

A final, small, subset of classes within RuggedJ are hand-written and loaded unmodified into the Java VM. These are classes that require specific, customized implementations within the RuggedJ network. For example, `java.lang.System` contains several methods for which we define special semantics: we must redirect all references to `System.out` to the head node, rather than to the local machine. Since performing such one-off transformations would be laborious and would complicate the transformation framework, we prefer instead simply to load a hand-coded version of these classes.

3.3 Method and Field Transformations

The implementation for most of the generated classes within RuggedJ follow simple templates: the stub and proxy classes each implement every method of the interface, with a standard bytecode sequence that performs a remote method invocation in the case of the stub, or forwards the method call to a referent in the case of the proxy. We optimize the

stub in some cases to cache immutable values, as we will discuss in Section 4.2.2. The remaining classes, `X_local` and `X_static_local` contain methods and fields copied from the original class. We rewrite the bodies of all copied methods to refer to the RuggedJ object model. This involves several rewrites:

Refer to new types. The first modification that we perform is to update copied method bodies (as well as copied fields) to refer to new types. In most cases this does not require a change. We type values by interface, and have designed our object model to re-use the original class' name as the interface name. However, there are some cases where we must update type names. As we will see in Section 3.4, we generate user-level equivalents for some system classes. In rewritten code we refer exclusively to these user-level types, and so we update any references in copied code. We also generate wrapper classes for arrays that make them conform to the RuggedJ object model. We similarly update references to arrays to correspond to these new types.

Call get and set methods. We generate get and set methods for the fields of each transformed class, allowing us to hide the location of these fields behind the interface. We rewrite the bytecode in copied method bodies to call these methods rather than directly access fields through `PutField` and `GetField` instructions. When calling these methods we take into account the different semantics of superclass methods and fields: methods override, while fields hide. A naïve implementation could access the wrong field if a subclass had a field of the same name and type. We avoid this by naming get and set methods with both the field name and the containing class.

Update method invocations. Since we type references by interface, we update method invocations from `InvokeVirtual` to `InvokeInterface`. The state of the stack required for these bytecodes is identical, so we need only change the operand. The exception to this rewrite is where we have declared a class to be Direct (see Section 3.4.3), and so do not indirect through an interface.

Convert array operations. Array operations pose some difficulty when rewriting. Unlike field instructions (such as `GetField`), array instructions (such as `AALoad`) do not en-

code the type of the array being operated upon (beyond whether it contains objects or primitives). The type of an object array's contents are determined at run-time based upon the contents of the stack, and so are not available to us when we rewrite the class. Since we wrap arrays we need to know the type of the content in order to call the correct get or set method. We determine this information through a simple data flow analysis that tracks the array type from its declaration. We use the same mechanism to convert `ArrayLength` bytecodes to a method invocation on the wrapper object.

Convert static references. Since we extract static data to static singletons, we also update any references to static methods and fields to use these singletons. This transformation is similar to the method and field rewriting described above, but with the minor complication that we must insert a reference to the static singleton before the call. This requires obtaining the reference (which we do through the static proxy class) and inserting it before any method parameters (which we pop to and then restore from local variables).

Convert static methods to instance methods. We transform the bodies of static methods themselves to account for their change to instance methods. Instance methods contain a reference to the containing object in their local variable slot zero, while static methods have no containing object, and so do not require this reference. When converting from static to instance, we increment the target slot for all local variable accesses by one, creating space for the **`this`** pointer. This could cause issues with offsets in the bytecode stream, since Java contains shorthand bytecodes to load to and store from low-numbered local variable slots. The toolkit we use to rewrite, ASM, bypasses this problem by abstracting away the shorthand bytecodes until it produces a final output sequence.

Rewrite monitor operations. Global synchronization in RuggedJ is handled in the library, and is discussed in Section 4.3.4. When rewriting method bodies we convert

all synchronization bytecodes (`MonitorEnter` and `MonitorExit`) to call out to the library, which ensures that they are executed correctly.

Wrap and unwrap references. We make extensive use of wrapping both for arrays and for system objects. When passing wrapped objects as arguments across the system boundary from user to system code, or when returning them in the opposite direction, we wrap or unwrap the reference to ensure that the correct object is seen on either side of the boundary. Passing from user to system code requires a simple unwrap operation to obtain the wrapped reference. Wrapping, on the other hand, requires that we check whether the object has been wrapped before to avoid creating two wrappers for a single object. We add a reference to the wrapper in system classes using the JVMTI agent (discussed in Section 3.4.2) which allows us to re-use existing wrappers. During the bytecode rewriting phase we identify those points where references pass from one side of the system boundary to the other, and perform compensating wrapping or unwrapping operations.

Add partitioning callbacks. We also make use of the rewriting process to install callbacks to the partitioning policy. These are discussed in more detail in Section 4.4; should a partitioning author wish to perform some action (such as object migration) on a trigger, the partitioning callback mechanism allows these triggers to be written into the rewritten code.

A final function of the rewriting phase is to replace allocation sites with references to our transformed classes. Allocation sites are the only occasion where we directly refer to generated classes, rather than to interfaces. Where the original application allocates an object of type `X` (using the `New` bytecode) the transformed version creates either an `X_local` or `X_stub` object, depending on the node upon which the allocation occurs, and a `X_proxy` object if the partitioning policy determines that the object may migrate. We can use `X_proxy`, `X_local`, and `X_stub` objects interchangeably in this manner because each implements the generated interface `X`. We make all method calls within rewritten code in terms of the interface, and field accesses go through the generated get and set methods. By

calling methods through interfaces, we minimize the transformation necessary on calling code, while maximizing flexibility in the types of objects used.

The decision whether to allocate an object locally or remotely, as well as whether to allocate a proxy, is made by the partitioning policy, and will be explored in Section 4.4. These decisions can be made statically (the classes to be allocated are hard-coded into the method bytecode), or dynamically (the partitioning policy is queried at whenever the allocation site is reached). The majority of allocations are performed statically, with local objects generated without proxies.

3.4 System Classes

The transformations described to this point apply only to user code, which can be rewritten by a user-defined class loader. The presence of system code within an application complicates the implementation of the RuggedJ object model. In this section we discuss the issues involved when handling system code, and present the transformations that allow us to integrate system code into our object model. In this chapter we focus only on the rewriting aspect of integrating system code. We defer consideration of semantics (such as migration) to our discussion of partitioning in Section 4.4.

We examine the restrictions imposed upon our system before we consider the impact of those constraints upon user code, because we find that system code is generally subject to more constraints than user code. Thus, as we will see in Section 3.6, the majority of constraints on transforming user code are caused by dependencies on system classes.

3.4.1 Barriers to Transformation

Java class loaders [Liang and Bracha, 1998] are organized hierarchically, as shown in Figure 3.8. The *bootstrap* class loader forms the root of a tree structure, with the *system* class loader as its only child. The bootstrap class loader is implemented within the Java VM, while the system class loader can be replaced with a user-defined class loader when the VM starts up. Any other user-defined class loaders form a tree rooted at the system

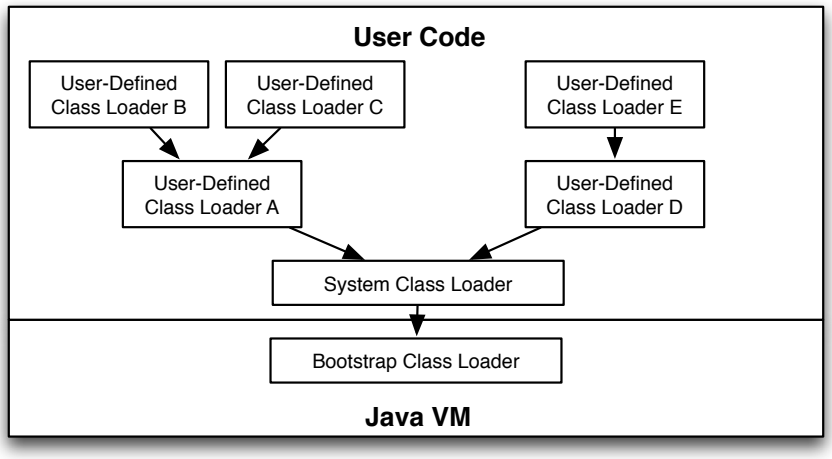


Figure 3.8.: Classloading in the Java Virtual Machine

class loader. A class loading request can explicitly specify the class loader by which it is to be resolved (using the reflective `ClassLoader` class). When the class loader is not explicitly specified, the class is loaded by the class loader responsible for the invoking class. By default, class loaders delegate all class loading requests to their parent in the tree. Thus, a class requested from User-Defined Class Loader E in Figure 3.8 would be passed through each parent node in the tree to be resolved by the bootstrap class loader. Should the bootstrap class loader fail to resolve the class then the request would be passed back to the system class loader, and so on. If none of the class loaders on the path through the hierarchy can load the class, a `ClassNotFoundException` is thrown.

RuggedJ's transforming class loader is loaded into the VM at boot time as the system class loader. Thus, any class loading requests that are not fulfilled by the bootstrap class loader are intercepted by our class loader, allowing us to rewrite all user code. Due to the complexity incurred by composing multiple user-defined class loaders, we do not allow applications to use custom class loaders.

This class loading structure poses two major problems for our transformation system. The first is that all system classes will be loaded by the bootstrap class loader, meaning that we do not have the opportunity to transform them. Further, while we could override

the delegation mechanism and transform the classes within RuggedJ's system class loader, Java's security mechanism would not allow us to load the transformed versions. User-defined class loaders may not load classes with reserved package names (such as `java.*`).

The second, and more fundamental, problem is that the class loading hierarchy imposes visibility constraints. A class can refer only to those classes loaded by the same class loader as itself or by a parent class loader. Thus, classes loaded by any user-defined class loader can refer to any system code (defined by the bootstrap class loader), but system code cannot refer to user code. This means that, even were we able to load transformed versions of system classes, they could not refer to user code such as the RuggedJ library.

A final barrier to transforming system classes is that some of these classes are effectively hard-wired into the VM. The bytecode that represents classes contains direct references to `java.lang.String` and `java.lang.Class`; both appear in the constant pool of a class file, and can be directly accessed using the `LDC` bytecode (that directly loads a constant to the stack). Again, changing the representation of these classes would require modifying the VM to understand the modified versions, which violates our goal for being able to run on any (unmodified) Java VM.

Interestingly, native and reflective code do not present any difficulty at the system level. Both types of code could break a system that transforms classes (and, indeed, must be accounted for within user code). However, since we do not rewrite system code, native and reflective operations perform as they would in an unmodified system.

3.4.2 The RuggedJ JVMTI Agent

The Java VM Tool Interface (JVMTI) specification [Sun Microsystems, Inc., c] provides a set of native interfaces that allow access to many aspects of the JVM's operation. It allows debuggers or profilers to interface with the VM. For example, an agent can extract performance metrics, or could monitor the threads in a running VM. Of interest to us is the bytecode modification functionality of the interface. There are two ways in which bytecode can be modified using the JVMTI: at class-load time and at run-time as a response to a

class rewriting event. Of the two, the former provides more flexibility. Run-time rewriting is subject to more constraints than load-time, as the modified code must be compatible with the running system.

While we cannot implement a custom class loader for system code, we are able to perform limited rewrites on the majority of system classes. By implementing a JVMTI agent we can intercept classes before they are loaded. However we cannot perform the full range of transformations on these classes. For example, we can only modify existing classes rather than generating multiple new classes. We do, however, make use of a JVMTI agent to perform some minor modification to certain system classes within the application. The implementation of some transformations, for example, is complicated by Java's access control mechanism; if we change the package to which a class belongs, we can no longer access other classes with default access in the original package. Our JVMTI agent modifies such classes to bypass these restrictions. Such a modification does not require reference to any additional classes, and does not alter program semantics, because the access control was checked statically at compile time.

Our JVMTI agent is implemented in C, using a custom bytecode modification library. The bytecode rewriting must be implemented in C; simply calling back to our ASM-based Java rewriting library would be tempting, but impractical. In order to rewrite a class this way would require loading of the entire ASM framework, along with the system classes upon which it depends. This would defeat the purpose of the agent, since it would miss rewriting hundreds of system classes before ASM had fully loaded.

The agent is called by the VM after a class is presented for loading by a class loader, but before it is actually loaded. We modify the class and return a new bytecode stream that is then loaded to the VM. This interface represents the major limitation to rewriting with JVMTI – we can modify classes but we cannot create or rename them.

A final limitation to class transformation is the presence of *primordial* classes. These are approximately seventy classes (with the exact number varying between VM implementations) that cannot be modified at all. Primordial classes are intimately tied to the VM, such as `java.lang.Object` or `java.lang.String`. Depending on the VM implemen-

tation, these classes may be hard-coded or directly memory-mapped to optimize startup times, and so cannot be intercepted.

3.4.3 Templates for Rewriting

Our strategy when handling system classes is to abstract away the distinction between system and user code, allowing rewritten code to refer to either without special cases. Thus we ensure that all system classes can be made to conform to the RuggedJ object model. Our class transformations use four basic techniques to obtain new types:

- The local instance of a *Wrapping* class holds a reference to a paired instance of the old type.
- *Extending* classes implement the object model through subtyping, with the generated local class extending the original system class.
- *Promotable* classes are not referenced by native code or by any other system classes, and so can be turned into user classes.
- *Direct* classes are not transformed, and so do not conform to the object model, solely because it does not make sense to for the target domain (the other transformations can be applied to such classes, but would result in unnecessary overhead). In a distributed system, immutable objects such as `Integer` need not be transformed, as they can be replicated on each node.

System Wrapping

Wrapping is the most straightforward of the transformation templates and is shown in Figure 3.9. In this approach, a set of classes are generated above the system boundary, in a special user-level package chosen to prevent name conflicts. For conciseness we refer to this package as `gen`. The base class is loaded by the bootstrap class loader, and is not modified. The local class contains new-type implementations of all the methods of the

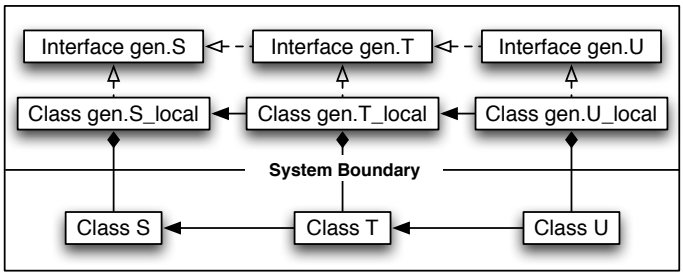


Figure 3.9.: Wrapping class hierarchy

base class, each of which translates the arguments from new to old, invokes the method on the wrapped base object, then performs an old-to-new translation on the return value if necessary. In this way a given object can be referred to by new type above the system boundary, and by old type below.

Unwrapping objects when passing from user to system code is a trivial operation. However, we must be more careful when performing the inverse; wrapping objects that are passed from system to user code. In this case we need to ensure that a given object that has previously been wrapped is reunited with its original wrapper; to do otherwise would create two wrappers for a single base object, which would not preserve identity. We avoid this by inserting a reference to the wrapper within each wrapped system class, along with get and set methods to access it. Since this involves the modification of system code, we perform this rewrite using the JVMTI agent. The wrapper reference is typed as `Object`, as a system class cannot refer to a user class. Finally, since we cannot add fields to primordial classes, we maintain a hash table for these objects, against which we check for existing wrappers before generating a new one.

As with all classes that conform to the RuggedJ object model, wrapping classes maintain the inheritance hierarchy of the original through their generated interface. That the local classes also subclass the relevant local class is merely a convenience—if they did not, every wrapper would have to implement redirect methods for the methods of every superclass, rather than just those in its base.

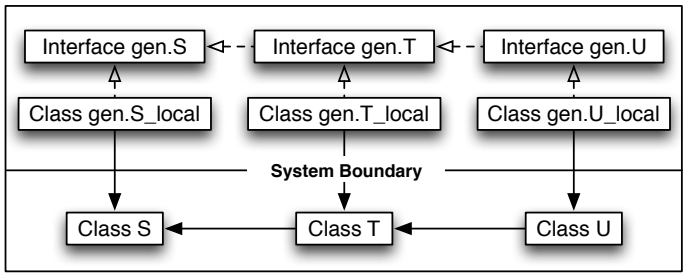


Figure 3.10.: Extending class hierarchy

The System Wrapping template can be considered the “universal solvent” for system classes. We can generate wrappers for any system class, which ensures that all objects in the application can conform to our object model. Unfortunately, the System Wrapping template also carries the highest overhead (as objects must be wrapped and unwrapped, which can be expensive), making the other templates more desirable.

System Extending

The System Extending template is an alternative means of handling system classes that eliminates the overhead of unwrapping. Under this technique, the generated local class extends the original base class, as shown in Figure 3.10. The generated interface and local class conform to our object model, while the base class remains unchanged. Note that in this case there is no inheritance relationship between the local classes; this is not important because the interfaces maintain the class hierarchy above the system boundary, while the base classes maintain it below.

An extending class can be passed to system or native code without any conversion process, since it extends the unmodified base. However we cannot create a new instance of an extending class within system code (as we cannot rewrite the allocation site to refer to `T_local` rather than `T`). This limits the applicability of this template to system classes that are only ever allocated above the system boundary. Further, while we obviously cannot extend **`final`** classes, we can also not override **`final`** methods. This may be an issue

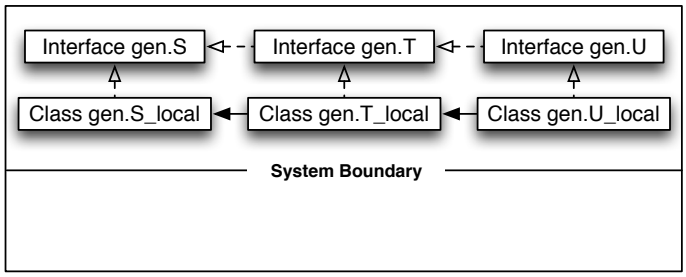


Figure 3.11.: Promotable class hierarchy

if a **`final`** method includes an old type as an argument or return value; the object model requires that such methods be overridden in order to be called by user code, which only uses new types. Thus, while the System Extending template is preferable to System Wrapping, due to its lower overhead it can be used only in limited cases.

Promotable

Promotable classes are a subset of system classes that are not referenced by any other non-Promotable system class or by native code. In this case we know that any reference to a Promotable class will either be in user code or in other Promotable classes. We can therefore move Promotable classes above the system boundary (by renaming their classes to form part of the `gen` package), and treat them as we do any other user class. Since we can rewrite all references to the Promotable class we can ensure that the original class is never referred to, and so is never loaded by the bootstrap class loader.

Promotable classes often exist in cliques within the system libraries, with no external uses from other classes in the libraries. An example that we have encountered is the Java XML processing library. If an application uses XML processing, much of the library is loaded into the VM. However these classes refer only to one another. Thus, we can *promote* these classes en-masse.

The structure of a Promotable class is shown in 3.11. This is the most straightforward implementation of the object model, with each local class implementing its interface.

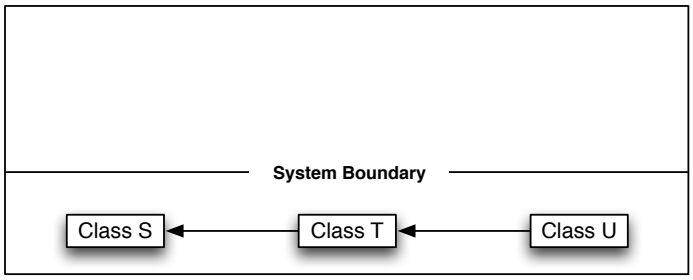


Figure 3.12.: Direct class hierarchy

While the inheritance hierarchy is maintained by generated interfaces, the local classes retain the original relationship. In the System Wrapping and Extending templates the actual method implementations were located in the base classes, Promotable local classes contain complete implementations of all their methods. Thus, Promotable classes must extend their parent so as to have their parent's methods available.

The Promotable template is similar to the Twin Class Hierarchy (TCH) approach proposed by [Factor et al., 2004], in that it loads system classes into the user space in order to perform transformations. However there is one important difference: the TCH system allows both modified and unmodified versions of the code to exist within a VM. We promote only those classes that are not used by other system code, so the promoted version is the only one in the system.

System Direct

The final set of classes, System Direct, do not conform to the RuggedJ object model. This template exists as an optimization; as we have seen, any class can conform to the object model through the System Wrapping template. However there are classes for which it is not necessary to conform to the object model. For example, when distributing an application with RuggedJ, we do not want to transform immutable objects. If we know that an object will never change, we can replicate it on multiple nodes, and eliminate the

Table 3.1: Subclassing between templates

	Wrapping	Extending	Promotable	Direct
Wrapping	Can subclass	Cannot wrap superclass	Cannot wrap superclass	No interface
Extending	Cannot alter base hierarchy	Can subclass	Cannot alter base hierarchy	No interface
Promotable	Cannot extend wrapper	Can subclass	Can subclass	No interface
Direct	No interface	No interface	No interface	Can subclass

overhead of remote method calls. Similarly, there are classes that are closely tied to the individual VM (such as `java.lang.Class`) that do not make sense to reference remotely.

Those classes we designate to be System Direct are not transformed in any way (as shown in Figure 3.12). As such they do not incur any overheads, and can be freely passed between system and user code, as well as to native methods. However, since they do not conform to the RuggedJ object model, they cannot be modified to extend the original application's functionality.

3.4.4 Subtyping

Since all of the transformation templates described above rewrite classes differently, we cannot freely "mix and match" techniques between super- and subclasses. Each rewriting technique therefore imposes restrictions on the classes of its hierarchy. The relationships are shown in Table 3.1.

Since System Direct classes do not conform to the RuggedJ object model, we must ensure that they have only other Direct classes in their hierarchy. To do otherwise would violate our rule that inheritance is maintained through interfaces; a Direct class has no interface, and so cannot fit into this scheme.

Likewise, System Wrapping classes can have only other Wrapping classes in their hierarchies. A Wrapping class cannot extend an Extending or Promotable class in case it is returned to user code from system code. There would be no way to produce a new-type representation of the Extending or Promotable superclass. The argument as to why a Wrapping class can only be extended by other Wrapping classes is similar. An Extending class that extends a Wrapping class removes our ability to translate from an old type to a new. In the case of a Promotable subclass, the local class would have to subclass the Wrapping subclass (since a Promotable object does not have a base class). This relationship would be lost when the base class was unwrapped.

A System Extending class can extend only another Extending class, since the local class must directly extend the base, and we cannot change the superclass hierarchy of the base class. However an Extending class can act as the superclass for a Promotable class; the System Extending template does not require unwrapping, so a Promotable local class can extend a System Extending local class without any loss of information should the object be passed to system or native code. This further indicates the usefulness of the System Extending template over System Wrapping. Promotable classes offer more options when extending an application's functionality, and by increasing the number of Extending classes, we likewise increase the number of potentially Promotable classes.

Our discussion of subtyping must also consider the original interfaces implemented by classes (as opposed to those generated as part of the RuggedJ object model). We rewrite interfaces in much the same way as classes: user-level interfaces contain signatures using new types, while system-level interfaces contain old types. Thus, system-level interfaces must be System Direct (if they contain only primitive or Direct arguments and return values) or System Extending (if they contain Extending, Wrapping, or Promotable arguments).

3.4.5 Classification

We refer to the process by which templates are chosen for each class as *classification*. A given class's classification may be determined by its subclasses or its references from

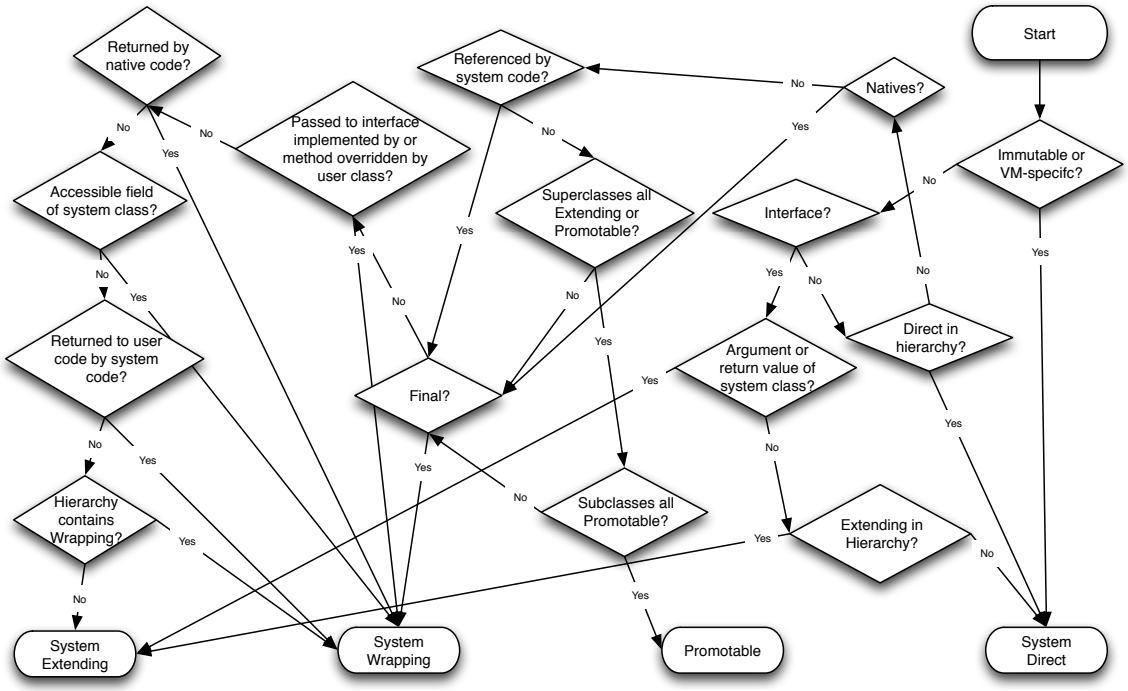


Figure 3.13.: Classification for system classes

elsewhere in the system, so we require knowledge of the entire application. We run the classification algorithm only on the classes that make up the application; analyzing the entire Java class libraries would introduce false dependencies, and limit our flexibility in transforming the application. We compute classification during a pre-processing phase, which we run once per application for a given set of class libraries.

We arrange the various classification templates using a total ordering. Direct classes are handled first, as they are an optimization and otherwise fall into at least one other classification. Next we find Promotable classes, which maximize the flexibility of our rewrites, then Extending classes that handle the remaining classes with less overhead. Wrapping classes account for the remainder.

The algorithm is iterative, since changes to the classification of one class may affect others. We present the algorithm as a decision graph, which produces the classification for a given class, assuming that all other classes have already been correctly classified. To

generate a full classification, we simply run the algorithm until a fixed point is reached. The decision graph for system classes is shown in Figure 3.13.

3.4.6 System Class Static Singletons

Extracting static members from a class is a simple process when transforming user code, but is not possible for system classes. Since we cannot rewrite system code, we cannot change static references (`InvokeStatic`, `GetStatic`, etc.) to use static singletons. Thus, we implement the static local class differently for user and system code.

We refer to the static local class generated for a user or Promotable class as a *mobile static singleton* (MSS). This local class functions as described in Section 3.2.5, and contains implementations of each static method from the original class, as well as versions of each static field. The methods and fields are transformed from static to instance members, allowing the singleton to implement the static interface. Additionally, by transforming static methods to instance methods we remove the dependency on a particular VM: static data is usually stored in a VM-specific manner and cannot easily be moved from node to node, while instance data is stored in the heap and can be migrated (hence *mobile* static singleton).

While system classes cannot use static singletons themselves, we make their static state available to remote user code by generating a *pinned static singleton*. This implements the same interface as a mobile static singleton, but can exist on only one node. The static local class contains an instance method with the signature of each static method. In this case it simply acts as a redirector, calling the static method of the system class, allowing that data to be accessed remotely.

Pinned static singletons pose a major barrier to distribution. Not only must all objects of a class with a pinned static singleton be allocated on the same node, but so must any other system classes that refer to the static parts of that class. Fortunately static singletons are required only for classes with static data. As we will see in Section 3.6.1, we do not

need static singletons in most cases; ultimately, only 12% on average of the classes we consider require a pinned static singleton.

3.5 User Classes

The transformation of system classes constrains that of user code. As we discussed in Section 3.4.4, the classification of a given type can affect the classification of its super- and sub-classes. This requirement extends above the system boundary, meaning that we need to create equivalent versions of the four templates within user classes. Additionally, native code can be present in user as well as system code, which limits our ability to rename and rewrite classes.

The four templates for rewriting user code closely mirror those for system code. Classes can be User Wrapping, User Extending, User Unconstrained (the user-level equivalent of Promotable), or User Direct. As might be expected, occurrences of user-level native code or the subclassing of system classes are rare. As we show in Section 3.6, the vast majority of user classes are either User Direct or User Unconstrained.

3.5.1 Rewriting

User code differs from system code in one important manner: the classes are loaded by our user-level class loader, and so can be rewritten. This has implications for User Direct classes, as well as the base classes for User Extending, and Wrapping.

When rewriting user code, we define two invariants:

1. Values with generated interfaces (User Wrapping, Extending or Unconstrained) are always typed using that interface. This allows us to vary the implementations of these interfaces among several alternatives (as discussed in Section 3.2). If we know that these instances will always be manipulated through the interface methods then any implementation of those interfaces is safely encapsulated and we can freely decide on that implementation without worrying if that decision impacts other code.

2. User code exclusively refers to new types. By strictly ensuring that all rewritten code uses new types, we define a clear separation between old and new types. We can maintain this invariant because instances cross the system boundary in well-defined places (passed as arguments, returned from methods, etc.). Thus, we never need to check dynamically if an instance is of an old or new type; the context from which the instance is referenced (system or user) decides statically if the instance has an old or new type.

We occasionally break the second invariant to optimize base classes. However, these violations are always localized transformations (an old-type reference never escapes the method in which it is used), and so do not impact the system as a whole.

3.5.2 Native and Reflective code

When transforming user code, we must make allowances for native code (for which we do not assume that we have source code) and for reflective code. We observe that either native or reflective code can break any large-scale series of transformations by introspecting on any class in the system. Should a class, field, or method be renamed or removed, hard-wired assumptions in native or reflective code may fail. We accept that an adversarial programmer, or one that makes extensive use of such code, can disrupt our system. We focus instead on permitting the widest possible range of common usages of both native and reflective code.

In the case of reflection, we do this at by intercepting reflective methods that refer to rewritten code and converting the results to the appropriate new types. We will discuss this mechanism in Section 4.3.2. In the case of native code, we exploit the heuristics laid down in J-Orchestra [Tilevich and Smaragdakis, 2006] that determine which classes are most likely to be accessed by native code. They define classes with native methods to be *unmodifiable*, as well as the types of their fields and superclasses (dynamic dispatch can result in calling an overridden method indirectly from native code). These heuristics are adequate for the applications we consider. We ensure that any classes that are likely to be

exposed to native code conform to the User Direct, Wrapping or Extending templates. This way they retain a base object upon which native code can operate.

3.5.3 Base Classes

User Wrapping and Extending classes are largely similar to their System equivalents, with the difference that their base classes are above the system boundary and so can be rewritten. Following the second invariant, we rewrite the method signatures and bodies of the base class to use new types rather than old. This simplifies the local classes that wrap or extend the base, since they do not have to translate between old and new types.

However, since user-level base classes may be passed to natives or system code (typed as system-level interfaces or superclasses), a base class must retain the *signature* of its unmodified original. New fields and methods may be added and the bodies of methods may be rewritten, but the class cannot be renamed, and its fields and methods must retain their original names and types. This violates our second invariant, that user code exclusively refer to new types.

We overcome this for methods by providing old-type implementations that simply redirect to their new-type equivalents. For fields this is more difficult. We ensure that any field that may be accessed by native code is not classified as User Unconstrained by the definition of unmodifiable classes above; a field of an unmodifiable class is itself considered unmodifiable, and so can not be classified as User Unconstrained. We observe that system code cannot directly access the fields of user classes, since they are loaded by different class loaders. Of the remaining templates, Direct and Extending classes are trivially compatible with system and native code (although we must type Extending classes as their base, and then cast upon use in user code). User Wrapping classes are also typed by their base, but since the wrapper is a separate object, we maintain a cached copy of the wrapper as an additional field. System or native code use the base class, while user code uses the new wrapper field. Note that the casting and wrapping of fields is required only in the base class

itself; all other user classes refer to the object by interface and so can never access the field directly.

Another violation of our invariant occurs when a method accesses its **`this`** pointer. The type of the **`this`** pointer in a base class is an old type. We must therefore convert the reference to a new type, either by casting if it is a User Extending class or by wrapping if it is User Wrapping. This way the invariant is maintained. There are, however, some situations in which this is not desirable and some in which it is not allowed. If the **`this`** reference is loaded to the stack in order to execute a field access, for example, we would rather perform the access directly rather than going through the `get` method of the interface. More importantly, if the pointer is loaded in preparation for a superclass constructor call (as required in every constructor) it would be incorrect to wrap the reference. Doing so would lead to the constructor being called on the wrapper rather than the base, which would cause a run-time error.

We determine which **`this`** references to convert using a def-use analysis. If the reference escapes the current method (by being passed as an argument or stored as a field) we convert it, otherwise we do not. While this violates our invariant that rewritten code exclusively refers to new types, it does so only in a localized manner. Note that we can also use this optimization when accessing local fields within Unconstrained classes.

3.5.4 Classification

The classification of user code follows a similar approach to that of system classes. Figure 3.14 shows the decision graph for user classes. The user classification process uses the same ordering as the system; Direct classes are handled first, then Unconstrained, Extending, and Wrapping.

3.6 Classification Evaluation

We evaluate our classification system using experimental results obtained from RuggedJ. We examine the output of our classification algorithm on a variety of benchmark applica-

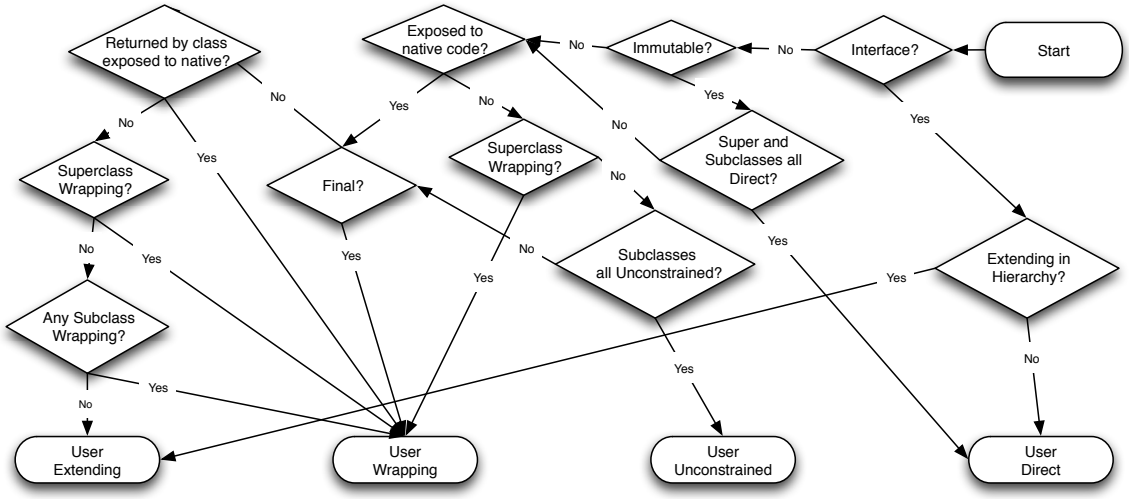


Figure 3.14.: Classification of user classes

tions, and provide some insight into the sources of overhead introduced by our system. Note that we evaluate classification on several standard benchmarks that we will not discuss in our overall performance evaluation in Section 5.3. The majority of standard benchmarks are unsuitable for distribution, either through low levels of concurrency or through bottlenecks in data organization. However they serve to illustrate the distribution of the various classifications.

All classifications were generated on an Apple computer, using Mac OS X 10.5.6, and version 1.6.0_07 of Apple's Hotspot-based Java VM. This affects the results of the classification; different implementations of the standard class libraries may produce slightly different classifications.

We ran the classification algorithm on applications from different benchmark suites, show in Figure 3.1: ten benchmarks from the the DaCapo suite (version 2006-10-MR2 [Blackburn et al., 2006]), nine from the SPECjvm2008 suite [SPECjvm98, 2008], plus SPECjbb2005 [SPECjbb2005, 2005]. In addition, we analyzed four of the applications that we will present in Chapter 5.3: a re-implemented distributable version of the SPECjbb2005 workload, a DNA database matching application [Keane and Naughton, 2005] and dis-

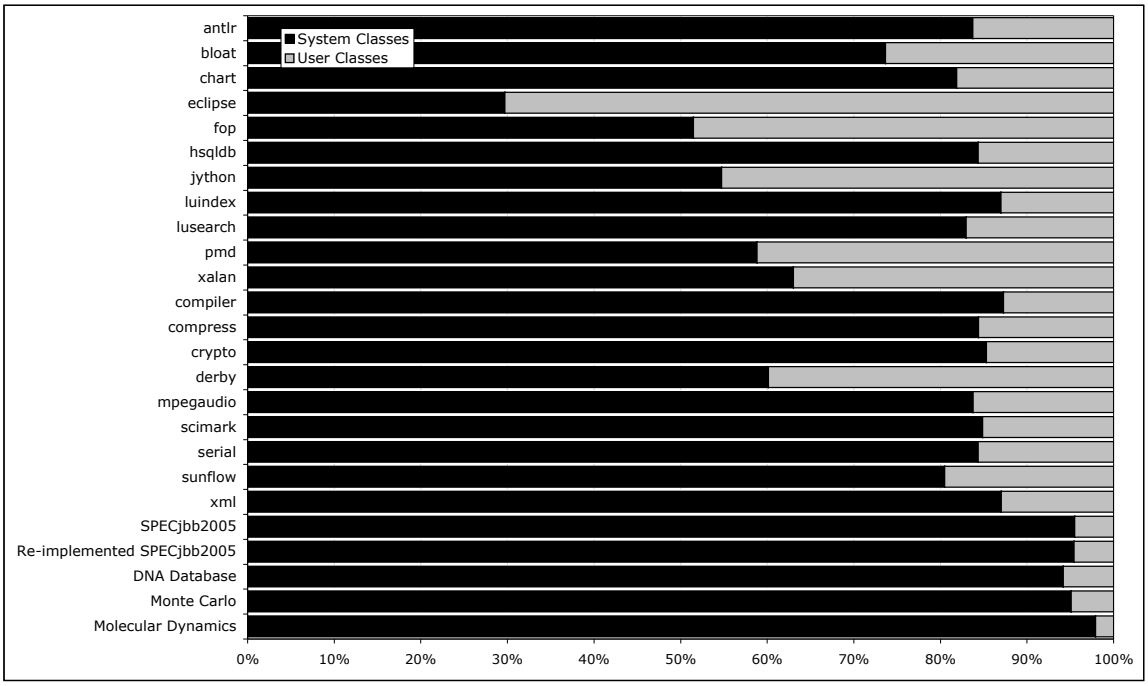


Figure 3.15.: Percentages of system vs. user classes

tributable versions of the Monte Carlo and Molecular Dynamic benchmarks from the Java Grande suite [Mathew et al., 1999, The Java Grande Forum].

To obtain an accurate count of the classes referred to by the DaCapo applications, we analyzed them without the DaCapo harness. This way we classified only those classes referred to by the application, not by the harness.

As Figure 3.1 shows, the majority of classes (78% on average) in an application belong to the standard libraries. This is due to the degree of interaction between system classes: a single reference can cause a large closure of classes to load. This strongly demonstrates the need to handle system classes within a rewriting system.

Figure 3.16 shows that the majority of user classes are split between User Direct (42% of user classes and 10% of the total application) and User Unconstrained (53% of user classes 13% of the total application). Very few classes are User Extending or User Wrapping. There was no user-level native code in the applications we studied, so these two classifications were used only for user classes that extended system classes. We see that

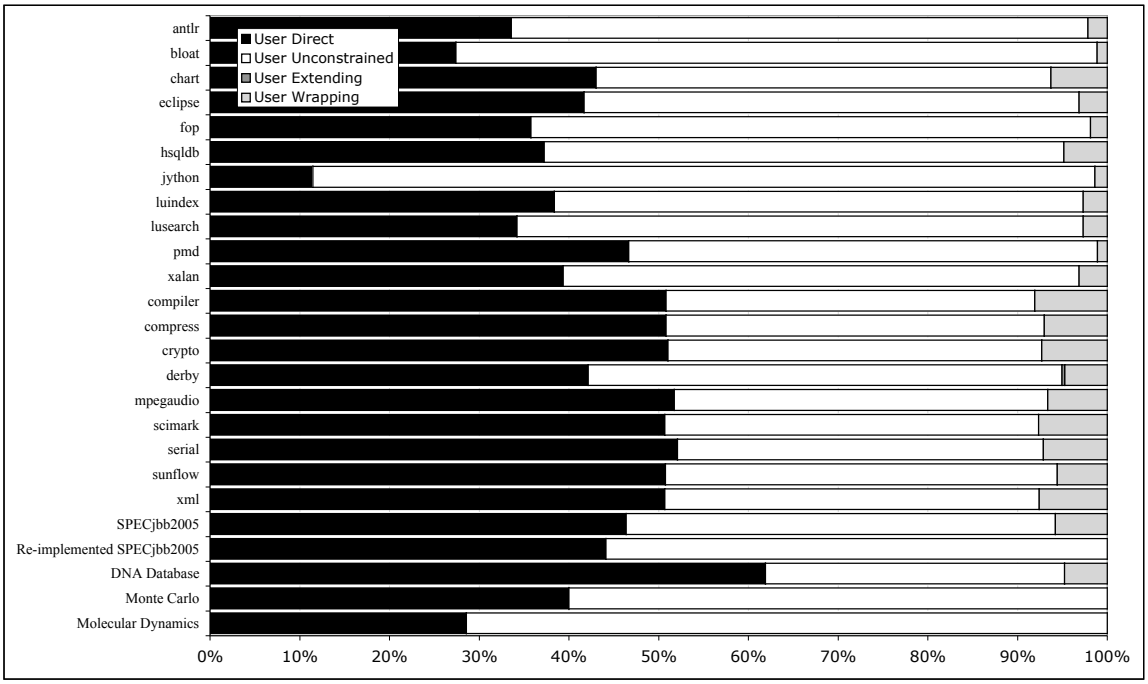


Figure 3.16.: Classification of user classes

only four classes in any of the benchmarks were classified as User Extending. While the number of User Extending classes seems insignificant, we must retain the classification template for these classes. Recall that an Extending class cannot extend a Wrapping class, so eliminating the User Extending template causes more system classes to be Wrapping rather than Extending, which we wish to avoid due to the wrapping overhead.

Figure 3.17 shows that, below the system boundary, System Wrapping classes are the most common, representing 57% of the system classes and 42% of the total application on average. This can be attributed to the need to wrap objects that are passed or returned to user code. System Extending classes are less common, representing 18% of system classes, while 20% of system classes are System Direct. Finally, 3% of classes on average can be promoted.

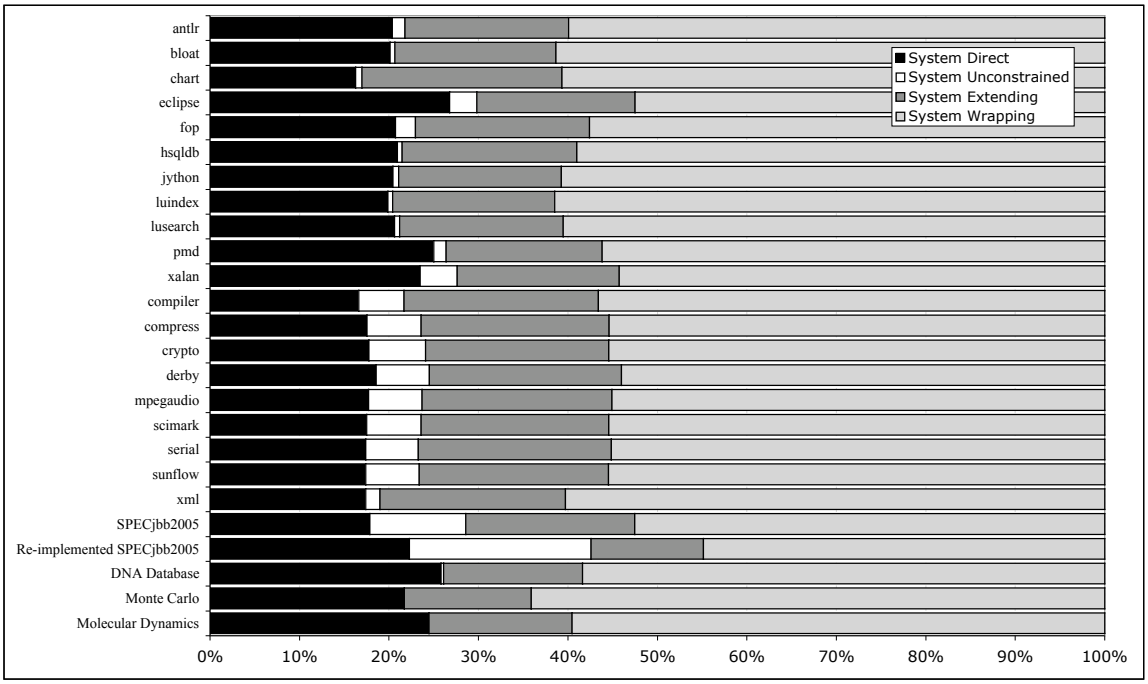


Figure 3.17.: Classification of system classes

3.6.1 Static Singletons

While we present a mechanism for handling static data within a distributed system in Sections 3.2.5 and 3.4.6, we recognize that static singletons pose a major source of overhead. At best, a local static method invocation requires additional work to locate and insert a reference to the static singleton, while at worst every static method invocation could become a remote call. We aim, therefore, to eliminate static singletons in those cases where they are not strictly necessary (for classes that have no static state, or for which static state is immutable). Figure 3.18 shows that we are able to completely eliminate static singletons for 82% of classes on average across our applications, 12% of classes require a pinned static singleton and the remaining 6% of classes need a mobile static singleton.

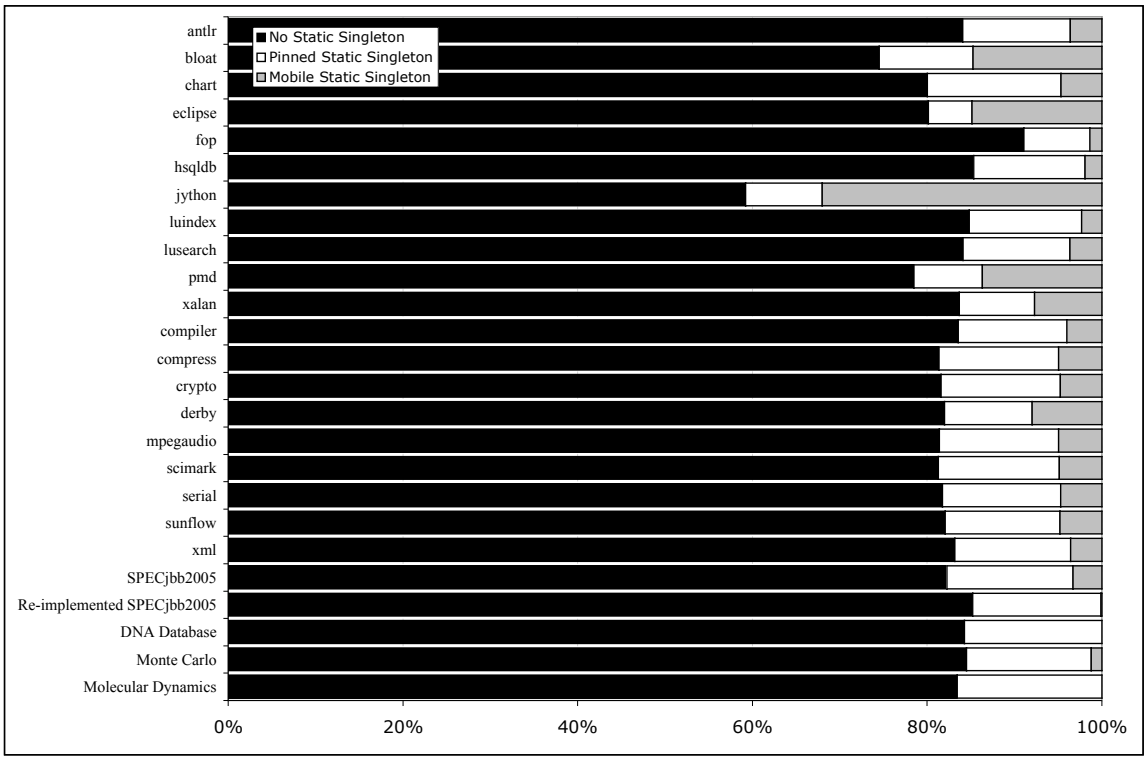


Figure 3.18.: Elimination of static singletons

3.7 Contributions

The RuggedJ object model borrows concepts from several of the systems presented in Section 2. The master/proxy model that we implement in our stub and local classes is a common theme in distributed systems; the majority of systems that refer to remote objects do so through some form of delegating stub. Where we differ is in the flexibility of our interface-based model. We type references as interfaces, allowing us to switch the implementation of an object from local to stub in the case of migration. We combine this ability with our dedicated proxy object that allows us to dynamically change the implementation of an object at run time, while retaining the ability to elide proxies for performance reasons when we know them to be unnecessary. No other object model that we studied has this flexibility.

Of the systems that we examined, Addistant had the most similar approach to handling system classes. Their Extending template was similar to ours, while their Copy template was equivalent to our Direct. However we allow more flexibility when placing system objects; Addistant clusters objects together by class, with all instances of a given class colocated, while we allow system objects to be allocated on different nodes so long as they do not refer to one another. Our whole-program rewriting system is also more flexible than those that we have seen. We allow programmers to override static classification decisions within an application's partitioning policy to take account of domain-specific knowledge. Thus we can declare objects to be functionally immutable (in a similar manner to Emerald, although within the context of Java) allowing them to be replicated. Similarly we can eliminate rewrites completely on key performance-critical classes, allowing them to run without the overhead from our rewrites; we have not seen this optimization performed by other systems.

3.8 Concluding Remarks

Bytecode transformation allows RuggedJ to integrate original application code with our infrastructure. In this chapter we have described a series of virtualizing transformations that can be applied to the various classes that comprise an application to allow them to conform to a unified object model, as well as the classification process to determine which template should be applied to each class. We have discussed the bytecode transformations that we apply to method bodies to integrate with this new object model, as well as the supplementary classes we generate to handle system code. Finally, we have described the implementation of our prototype system, and presented an analysis of the classification process when applied to several benchmarks.

4 RUN-TIME SUPPORT

RuggedJ's run-time system links the rewritten classes around the network, creating a single unified application. It perform the dynamic aspects of distribution; where the bytecode rewriting class loader transforms classes to handle remote objects, the run-time system creates, tracks and dispatches to those objects. It provides an abstract interface to library functionality, allowing complex tasks to be moved from rewritten bytecode to pure Java implementations, and it manages global operations such as the coordination of static data or synchronization between objects. Finally, the run-time can optimize an application without modification to the original source, improving data locality by migrating or replicating objects.

The RuggedJ run-time is made up of separate library instances running on each node in the network. The major functions of each node's run-time include:

Library functionality. Transformed code calls out to the run-time library to perform tasks that would be laborious to implement in bytecode. For example, when a wrapped object is returned from system code to user code it must be wrapped, re-using a previous wrapper object if one exists (as described in Section 3.4.3). This wrapping operation can take various forms, depending on the object: user code maintains a typed reference to previous wrappers, system code contains an object reference that must be cast, and primordial classes can not hold additional references, and so the wrapper must be located in a hash table. Finally if a wrapper does not already exist, a new wrapper object is reflectively created and then stored for future lookups. Rather than providing the bytecode sequence to perform these operations at every wrapping point we simply call out to the run-time library.

Network communication. When a RuggedJ network is created, the run-time systems on each node negotiate to form a communication hierarchy (discussed in Section 4.1.2.

This communication system is used to query network state information (such as the existence and location of static singletons) and to broadcast control messages (such as the global termination command). As the application executes, the run-time systems coordinate to create and invoke methods on remote objects.

Remote Object Tracking. The RuggedJ run-time tracks remote object references, both when external object references are introduced as arguments for incoming messages or when local references escape the node. Mobile object locations are updated using broadcast communication, and nodes redirect requests to moved data.

Replication and migration. Data locality is essential for performance in a distributed system. Since network accesses are orders of magnitude slower than local memory reads and writes, a very few key remote objects can devastate performance. Our run-time system supports object *migration*, allowing data to move to the node upon which it is to be used. Additionally we *replicate* immutable data on each node in the network, allowing local read-only access to those parts of an object that never change.

Threading and synchronization. The run-time system preserves the identity of threads in the original application by mapping each logical global thread to a specific Java thread on each node. This *thread affinity* ensures that application behavior that relies on thread identity, such as the acquisition of monitors, operates correctly in the distributed application.

The final component of the run-time system is the partitioning strategy. This determines how the application is divided between the RuggedJ network's nodes, and controls the location and migration of objects. Partitioning in RuggedJ is performed by an application-specific plug-in written by the application developer. This allows the programmer to leverage domain-specific knowledge of the application's structure and behavior, while benefitting from the powerful tools provided by RuggedJ.

The remainder of this chapter is structured as follows: Section 4.1 describes the RuggedJ network, both in terms of architecture and specification. Section 4.2 discusses the primitive components upon which RuggedJ's run-time system is built, and Section 4.3 describes how

we compose these primitives to implement some of the Java language's features. Finally, Section 4.4 describes the design and capabilities of RuggedJ's partitioning interface.

4.1 The RuggedJ Network

RuggedJ is designed such that the configuration of the network is supplied as a final step in the deployment of the system. A given application (with a well-written partitioning strategy) can be executed on an arbitrary network without special modification. In this section we describe the specification and configuration of networks within RuggedJ, as well as the mechanism by which we communicate between nodes.

4.1.1 Network Configuration

RuggedJ requires that all nodes in a network are capable of running a fully-featured Java virtual machine, and that each node implements the same version of the standard class libraries. Differences between library versions may lead to incompatible code executing on different nodes. Beyond this requirement, RuggedJ is agnostic to the virtual machine implementations upon which it runs. A single RuggedJ network may run on nodes with different architectures, operating systems and virtual machine implementations.

Nodes within a RuggedJ network communicate over Java sockets; we found simple Java sockets to perform better than MPJ [Baker and Carpenter, 2000], a Java implementation of the MPI specification that we had used through much of the development of RuggedJ. At present network configuration is supplied as a text file when the network is created, although future versions may detect peer nodes automatically. The network configuration specifies the host name for each virtual machine, as well as the port upon which it listens. This way a single host may run multiple RuggedJ nodes. When each node starts up it ensures that it can reach all other nodes in the network and that all nodes are running a compatible virtual machine. This guarantees that the network is ready before we begin execution of the application.

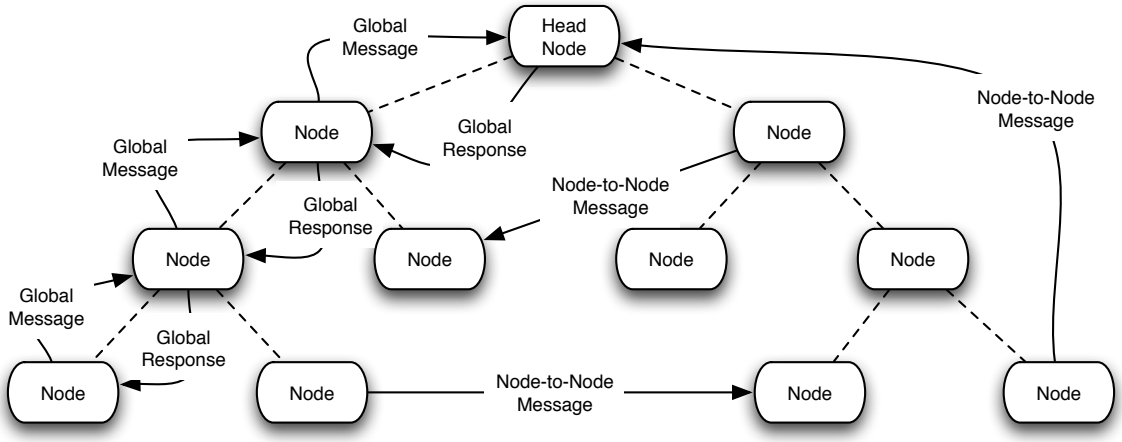


Figure 4.1.: Communication between RuggedJ nodes

4.1.2 Communication

When each node has determined that the others in the network are valid and reachable, the nodes organize themselves into a binary tree structure, establishing parent and child relationships with other nodes in the network (Figure 4.1). The hierarchical structure is determined dynamically by the nodes in the network, and is not supplied as part of the network configuration. This simplifies the configuration of the network, and leaves opportunities to tailor the communication hierarchy based upon run-time network-specific factors such as host location and connection speed. By arranging the nodes in this manner we simplify the dissemination of run-time information across the nodes; a node need only exchange run-time information with its parent and children rather than broadcasting to the network as a whole.

The node at the root of this tree is designated the *head node*. This node has several responsibilities beyond those of a regular node. First, it instantiates the application by invoking the `main` method on the appropriate application class. Additionally, the head node serves as a canonical source for global information, and arbitrates global activities (such as the allocation of static singletons, discussed in Section 4.3.3). Since the head node sits at the root of the communication hierarchy, we can be certain that any requests for global

information passed up the tree will eventually be fulfilled. Finally, the head node manages the standard input and output streams, displaying all application output to a single console.

Communication within RuggedJ is performed by passing `Message` objects between nodes. Messages are split into two categories: *global* messages that concern the status of the network as a whole, and *node-to-node* messages that communicate between arbitrary nodes. Global messages include class loading requests that ensure the uniqueness of static data, queries as to the status of other nodes, or a shutdown message when the main method exits or in response to a `System.exit` invocation. Global messages make use of the communication hierarchy; updates and queries pass up the tree to the head node, while responses and relevant information passes back down from parent to child. Most messages within the system are node-to-node messages. These include remote method invocations, object replication and migration, and requests to allocate objects remotely. These messages are passed directly between the nodes concerned, without involving the head node.

4.2 Run-Time Primitives

The RuggedJ run-time system provides the functionality that we need to correctly distribute an application. In this section, we discuss these primitives, while in Section 4.3 we describe how we exploit them to support Java's semantics.

4.2.1 Object Management

Objects within RuggedJ do not have a direct one-to-one relationship to those within the original application. As discussed in Section 3.2, the implementation of an object depends on whether it is local or remote to the current node, and whether it may migrate (in which case it requires a proxy). The RuggedJ run-time library creates and maintains these various representations.

Any object that may be referred to from a remote node is given a unique global identifier (*UID*). UIDs are created on-demand when an object reference first escapes a node; we do not need to create UIDs for purely local objects. A UID comprises a **`long`** (the encoding

is based on the creating node's network identifier so UIDs can be created locally) and a `boolean` indicating whether it requires a proxy. Whenever an object reference is passed between nodes (as part of a method invocation, for example) we refer to the object by UID.

UID references are resolved immediately upon reception by a node. This way we know that all references within rewritten bytecode correspond to the RuggedJ object model, and do not have to check for UIDs. When a UID is encountered in an incoming message, the node's run-time system first checks whether this UID has been seen before. Each node maintains a map of all UIDs that it has sent or received to their representative local or stub instance on the current node. This way we ensure that multiple resolutions of a given UID result in the same instance. Should an incoming UID not be found in the map, we know that the object to which the UID refers has not yet been seen by the current node. This implies that the object is remote; a local object's UID is logged in the map when a reference to it first escapes the node. We therefore create a stub (and proxy if necessary) to represent the object on this node.

4.2.2 Immutable Objects

By far, the largest source of overhead in RuggedJ stems from accessing remote data. Each remote field access, as well as method invocations on remote objects, requires the creation and delivery of a network message, along with the associated marshaling and unmarshaling of target objects, arguments and return values. We have found that the cost of sending a message overwhelms the cost of transferring data. Thus, we can improve performance of the system as a whole by reducing the number or remote object accesses, even if it means transferring more data per message than would otherwise be the case. The simplest way to do this is to exploit immutability. If an object is known to be immutable (i.e., that its fields are never modified after initialization) we can safely replicate the object on any node that refers to it. The state of these copies will never change, meaning that no coherence mechanism is needed to keep them up to date.

We identify immutable objects at the class level using a whole-program analysis, examining the bytecode of every method in the transitive closure of the application to find occurrences of the `SetField` bytecode. We could have decided immutability simply by using the **`final`** keyword, but it is common even that non-**`final`** fields can be statically inferred immutable, so our our analysis produces a larger set of immutable classes. The static immutability analysis comes at minimal marginal cost, since we must statically analyze the whole program anyway to determine interactions between user and system code (discussed in Section 3.13). We do not generate local, stub or proxy classes for immutable classes (a replicated class does not need a stub, since it will never be referenced remotely).

Replicas of immutable objects are created in the same way as stubs for remote objects. Each immutable object has a global **`long`** identifier (ID) similar to a UID. As an optimization we do not use actual UIDs for immutable objects because they never need a proxy and so there is no need to track that information. When an immutable ID is received, a node first checks whether that immutable object has been seen before, and if not it creates a replica.

Replicas are created in one of two ways. If the object to be replicated implements `Serializable`, we use Java's serialization mechanism to transfer its data, constructing a **`byte`** array that can be sent across the network. Otherwise we must use reflection to extract each of its fields, and send a map of field/value pairs that allow the object to be reconstructed reflectively at the destination. Of these approaches, Java's serialization is preferable; the serialization mechanism is considerably faster than reflection over fields. However serialization is appropriate only when an object's transitive closure can be serialized. The object cannot contain fields that should be transmitted as UIDs, otherwise serialization will erroneously make a copy of the stub or local object. Thus, serialization is limited to objects with primitive or immutable fields.

In addition to immutable objects, certain parts of otherwise mutable objects can be replicated. Individual fields within a mutable class may themselves be immutable; we can cache these values on each node, meaning that we do not need to make repeated network calls to read the data.

We implement this caching within the stub class. During the static analysis we determine the *immutable content* of each class. When we generate the stub for a given class, we include state for its immutable content. We supplement the generated `get`/`set` methods to check whether the immutable content has been loaded, and if not we request the immutable content from the original object. Caching immutable content on demand rather than when the stub is created at a node avoids transferring state that is never accessed. However, all the immutable state for an object is requested at once to avoid multiple requests for different immutable fields. These decisions minimize the number of requests for immutable state.

We further observe that we need only execute methods on the node containing an object if that method contains synchronization or modifies the object's fields. Thus, we locate unsynchronized methods that do not access any of the mutable fields of the target object and replicate their bodies in the stub class. We can execute these methods on the local node without the overhead of a remote method invocation.

4.2.3 Migration

Data locality is key within any distributed application. Manipulating remote objects requires a network transaction, which is more expensive than local accesses by orders of magnitude. Therefore, we minimize non-local accesses by *migrating* objects between nodes. Migration can exploit shifting spatial locality; a good example is the allocation of a large array on one node A, followed by its use on a different node B. Without migration, one must either create the array on A and perform remote accesses from B, or, conversely, create it on B and populate it from A using remote accesses. Either case suffers significant performance degradation. Instead we allocate and populate the array on node A, then migrate it to node B, requiring only one network access. The cost of sending a message, (the message creation time, and the time taken to marshal and unmarshal its arguments) greatly outweighs the time taken to actually transfer the data across a fast network connection. Therefore, migrating an object to cut down on remote method invocations, while involving

a single large data transfer, is significantly cheaper than repeated remote invocations on that object.

The run-time system must determine which objects can migrate. A migratable object must have a proxy in order to redirect references from the formerly local object to its stub. We determine which objects require a proxy by querying the partitioning policy. Further, an object can be migrated only if all references to that object go through the proxy. This means that objects manipulated by either system or native code cannot migrate; such code is unaware of the RuggedJ object model, and so cannot refer to remote objects. Finally, we cannot migrate objects with close ties to the JVM (such as `java.lang.Class`); in practice migrating such objects would be meaningless. These constraints allow us to migrate the vast majority of user-level objects, as well as system-level objects referred to only from user code (such as utility objects drawn from the `java.util` collections framework), which generally make up the parts of an application that we would like to migrate.

Deciding which objects are to be migrated and when they should migrate is determined by the partitioning strategy. We provide tools that present developers with the results of the static partitioning analysis, determining which objects are referenced from which classes. However the constraints imposed by a static analysis are generally more stringent than those needed in practice; partition developers can frequently override these limitations to improve the performance of their applications.

The partitioning policy interface provides several mechanisms by which object migration can be triggered. The policy author can indicate certain methods whose results should be migrated. This gives a mechanism to implement the earlier example; a large array may be created, initialized, and returned by a given method, then migrated to the node upon which it will be manipulated. The run-time library also maintains a count of remote accesses to a given object. The partitioning policy can determine a threshold number of accesses before it is given the option to migrate an object. This can be useful if an object is alternately referenced by multiple nodes. It can migrate to the node currently making use of it, and then move to the next node after reaching the threshold. Finally, migration

can be triggered through user-defined callbacks to the partitioning strategy. These will be discussed further in Section 4.4.

Migration builds upon the same primitives as object replication. When the partitioning policy determines that an object should migrate, both the source and destination nodes are notified. We update the proxy reference on the source node to refer to a newly-created stub object. Any new accesses to the object during the migration are forwarded to the destination node, which will block them until the migration is complete. The proxy maintains a count of threads entering each method (including generated `get`/`set` methods), and so can wait until all outstanding invocations on the object have completed. This has the potential to create deadlock if a thread recursively invokes methods on the same object; new method invocations will be forwarded to the remote node and never return. This must be avoided by taking appropriate care when defining a partitioning policy, although we have not encountered such a situation in the applications that we have distributed.

Once all threads have exited all methods on the migrating object, the local object is copied using the same mechanism as in object replication. When the copy is complete, the source node forwards any waiting invocations to the destination. Finally, the new location information is transmitted to the head node, from where it propagates to all nodes in the network. Subsequent invocations received by the source node from third-party nodes are forwarded to the destination, and the third party informed of the change.

4.3 Java Semantics

Preserving source-level Java semantics in a distributed context is a challenge, and requires explicit run-time support to ensure *object identity*, appropriate reflective behavior, uniqueness of static state, proper thread synchronization and exception handling.

4.3.1 Object Identity

Since an original application object can be represented by multiple generated objects within RuggedJ (local, proxies and stubs), we must maintain object identity. Comparisons

between objects (e.g., == and `.equals()`) must produce the same result in the distributed version of the application as in the original. The design of the RuggedJ system ensures this property with no additional effort.

Consider mutable objects that conform to the RuggedJ object model. Method-based comparisons will trivially produce the correct outcome: any method invocation upon a remote object is forwarded to that object's local instance on the remote node, where it executes like any other method. Of more interest is the == operator. The map from UID to instances on the current node (as discussed in Section 4.2.1) ensures correct execution of ==. We cannot have a stub and a local version of the same object on the same node, and we cannot have two stubs referring to the same remote object on the same node. Additionally, a local or stub object for which a proxy is created is never referenced without that proxy. These properties, plus the uniqueness of each local object (by definition) ensures that == comparison simply works. Any two references that represent the same UID object on a given node will be reference identical.

The argument for immutable replicated objects is similar. We ensure that only one copy of an immutable object is created per node by maintaining a map of all previously encountered immutable identifiers. Since a direct object is uniquely represented by a single instance on each node, comparison methods and reference identity (==) comparisons produce the proper results.

4.3.2 Reflection

Reflection allows Java application code to introspect on itself and to execute arbitrary methods. The difficulty that this poses for any dynamic rewriting system is clear; if an application should reflectively access code which has been modified or removed, the effect on the system will be unpredictable or catastrophic. Fortunately, however, applications generally use reflection sparingly and RuggedJ is able to cope with the common uses which we have experienced.

Handling reflective code in RuggedJ involves both the bytecode rewriting class loader and the run-time system. During the rewriting process, we check for reflective calls, filtering by class name. For example, when we find a method invoked on a `java.lang.Class` object we handle it using our reflection mechanism. We rewrite the invocation to be a static call to our reflection manager within the run-time system; in the case of an instance method we pass the target object as the first argument to the static call, eliminating the need to modify the stack. In the run-time library we attempt to replicate the intent of the call within the context of our rewritten system. For example, should an application invoke `getMethod` on a class object, we return the result of that call on the local version. We have found this approach to work well in practice.

There is, however, a fundamental problem with such reflective transformations. An automated system cannot accurately know the intent of the developer, and so we may return the wrong result in some instances. For example, should an application call `Class.forName` to obtain a `Class` object, the developer may want the new type (e.g. a user-level wrapper class) to invoke a method, or may want the base type to obtain a lock on the class. In such cases our system may produce incorrect results. Developing a better system for handling reflective code could be an interesting research project in the future.

4.3.3 Static Data

When distributing a Java application we must retain the semantics of static data, ensuring that a static field maintains a single, global, value regardless of the node from which it is accessed. In RuggedJ we encapsulate static data using a per-class *static singleton* object that holds the canonical version of the static fields of the class.

The structure of a static singleton (shown in Figure 3.7) closely mirrors that of the transformed instance parts of a class (from Figure 3.3). This allows a static singleton to be remotely accessed in the same way as any other object and, subject to partitioning constraints, to migrate. The static singleton structure differs slightly from that shown in Figure 3.3 in that the static proxy does not implement the static interface. This is because

```
1 public class X_static_proxy {
2   private static StaticGeneratedClass singleton;
3   public static StaticGeneratedClass getSingleton()
4   {
5     if (singleton == null)
6       singleton = StaticSingletonManager.getSingleton("X");
7     return singleton;
8   }
9 }
```

Listing 4.1: Getting a static singleton via the static proxy

a static singleton does not represent an object from the original application, and can never have its reference stored. We use the static proxy as a means of obtaining the static singleton when necessary, but we do not cache it in rewritten code.

The classes of an application fall into three categories with respect to static data. First are those that contain no mutable static fields. In this case we have no need for a static singleton, since there is no static data to manage. For established benchmarks such as SPECjvm2008, SPECjbb2005, and DaCapo, approximately 82% of classes do not require a static singleton [McGachey et al., 2009b]. Of the remaining classes, we split static singletons into *mobile* and *pinned*. Mobile static singletons can be allocated on and referenced from any node in the network, while pinned singletons have data exposed to system code and so have constraints upon their locations. Untransformable system code referring to static data is a problem inherent to Java distribution, requiring that the application be partitioned in such a way as to avoid duplication of fields. We defer to the developer of the partitioning strategy for a correct partitioning, guided by our classification tools.

The RuggedJ run-time library tracks the locations of both mobile and pinned static singletons, and ensures that only one instance is created in the network. Rewritten code obtains a reference to a given static singleton through its static proxy, as shown in Listing 4.1. The `StaticSingletonManager` class coordinates with its counterparts on various remote

nodes to obtain a reference to the static singleton, or to create one if it does not already exist. It first sends a *class loading query* to its parent in the network to find the singleton. If the parent has a reference it returns it, otherwise the request continues up the tree until it reaches the head node. If the head node does not have a reference to the singleton it assigns the original requester to create one (logging the fact to prevent race conditions should two nodes simultaneously request a singleton for the same class). This way, the head node maintains a record of the location of all singletons, and can redirect future class loading queries to the correct node.

4.3.4 Threading and Synchronization

RuggedJ distributes applications by transferring thread control flow between nodes. This generally occurs by invoking a remote method; the source node waits while the destination executes the method, returning control to the source when the method completes. In this way we aim to execute a method on the same node as its data, rather than bringing the data to the appropriate node and suffering the overhead of object migration. A naïve approach to remote method invocation would be to maintain a pool of threads on each node, and use these threads to execute any incoming method requests. However doing so would violate Java's synchronization semantics.

Java **`synchronized`** blocks and methods express *monitor* synchronization with respect to an object instance or class [Gosling et al., 2005]. The monitor (instance or class) is named explicitly in **`synchronized`** blocks. In **`synchronized`** methods, the monitor is implicitly the method receiver (i.e., **`this`**) for instance methods or the declaring class for static methods. At the bytecode level, **`synchronized`** blocks translate to blocks delimited by `MonitorEnter`/`MonitorExit` bytecodes. Only one thread at a time can acquire a monitor, though the same thread may recursively acquire the same monitor multiple times. This allows a thread to call multiple **`synchronized`** methods from within the same monitor without complication.

```
1  public class X {
2    private Y y;
3    public synchronized void m1(){
4      y.method(this);
5    }
6    public synchronized void m2(){
7      ...
8    }
9  }
10
11 public class Y {
12   public void method(X x){
13     x.m2();
14   }
15 }
```

Listing 4.2: Example: **`synchronized`** methods

Recursive monitors force threads executing remote **`synchronized`** methods somehow to retain the identity of their calling thread, even when the caller is on a different node. Consider the situation in Listing 4.2. A thread that invokes `x.m1()` obtains the monitor for the appropriate `X` instance, then calls `y.method(this)`. When `method` invokes `x.m2()` the thread must be able to re-enter the monitor for the `X` instance, and successfully execute `m2`.

However, under RuggedJ it is possible for the objects in this example to be distributed as shown in Figure 4.2. In that case, the thread on `Node A` enters the monitor for the appropriate `X_local` instance, then waits for the remote method invocation of `y.method(X)` to return. The thread on `Node B` makes a remote call back to `Node A`. If a random thread is assigned to execute this call then it will deadlock waiting to acquire the monitor on the `X_local` instance.

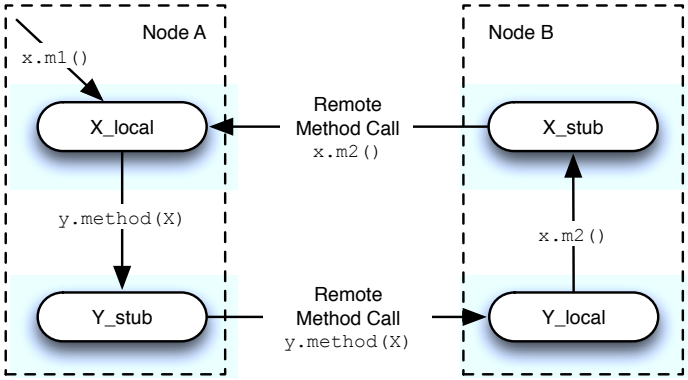


Figure 4.2.: Problem: **`synchronized`** deadlock

Thread Affinity

We solve this issue by implementing *thread affinity* for remote method calls. We define logical *global threads* that capture the control flow of a single thread in the original application. We map each global thread to exactly one Java thread on each node; any remote method call performed by a given global thread is executed by its assigned thread on that node. This ensures that the situation in Figure 4.2 cannot occur.

Since we have built our thread affinity implementation on top of our custom network library we do not rely upon Java RMI to assign threads to remote method invocations. Thus we can use a simpler thread affinity system than that used by J-Orchestra [Tilevich and Smaragdakis, 2004]. We can control which thread executes remote methods so we do not need to map threads to equivalence classes like J-Orchestra does, and we are able to use the JVM's internal monitor implementation rather than creating a custom synchronization library.

To minimize the number of Java threads running on each node we only create local threads on demand. If a global thread invokes a method on a given node for which it currently has no local thread assigned, then one will be created to carry out the work. We must also intercept remote invocations of `Thread.start()` to create a new global thread, and detect termination of a thread when its initial `run()` method exits. This triggers a clean-up process that frees the corresponding local thread resources on each node.

Implementation of thread affinity requires that method invocations be non-blocking; the sequence of calls in Figure 4.2 would lead to deadlock if the first remote invocation (of `y.method`) blocked, since the Java thread would be unavailable to execute `x.m2`. Instead, once a thread initiates a remote method call it proceeds to wait for incoming messages, allowing it to execute any incoming method calls.

Listing 4.3 sketches the implementation of our non-blocking calls: `invokeRemoteMethod` is called from a stub object, indicating that a running method has attempted to invoke a method on a non-local object. The stub creates an `InvokeMethodMessage` object that describes the method to be invoked, as well as its target object. After sending the invocation request in line 2, control passes to `waitForMessages`.

Each global thread has a unique thread identifier (TID) that is constant throughout the network. We use the TID to determine the thread for which a message is intended. When a message arrives for a thread (line 9 in Listing 4.3) we can tell from the type whether the message is a response to an outstanding remote call or a remote request for a new method to be executed locally. If the former, we simply return the value, and control proceeds from the site of the initial remote invocation. If the latter, we execute the incoming method on the local node (line 15). Any subsequent remote method invocations in the method body will result in a call to `invokeRemoteMethod`, which will call `waitForMessages`. Thus, we can see that successive remote invocations for a given thread form a stack of `waitForMessages` calls, ensuring that return values are properly matched to their respective invocations.

Remote Monitors.

When acquiring a monitor in a distributed context, we must ensure that the correct object is locked. In RuggedJ this requires two conditions: the monitor must consistently be acquired on a specific node, and the monitor must always be acquired on the same object within that node.

This second condition arises from the various implementations of the RuggedJ object model, outlined in Section 3.2. In order to make untransformable system code conform

```
1  public Object invokeRemoteMethod(InvokeMethodMessage msg) {
2    msg.send(thread_id);
3    Object result = waitForMessages();
4    return result;
5  }
6
7  public Object waitForMessages() {
8    for(;;) {
9      Message msg = Message.recv(thread_id);
10     if (msg instanceof MethodResultMessage) {
11       return ((MethodResultMessage)msg).getResult();
12     }
13     if (msg instanceof InvokeMethodMessage) {
14       Object result
15         = ((InvokeMethodMessage)msg).invokeLocally();
16       new MethodResultMessage(result).send(msg.source,
17         thread_id);
18     }
19   }
20 }
```

Listing 4.3: Non-blocking remote method invocation

to the object model we create user-level wrappers around unchanged system class objects. Thus we must ensure that user-level code locks the system object, rather than the wrapper. This way both user and system code acquire monitors on the *same* target object; user code by selecting the correct object, and system code which is not aware of the user-level wrapper.

Additionally, we optimize immutable objects within RuggedJ by replicating them across the network. For these we define a replicated object's *home node* as the node upon which it was first allocated, and use that copy as the canonical target for locking. Immutable objects

locked by untransformable system code are by default not replicated (though this can be overridden by the partitioning strategy), so they do not pose any problem.

We implement remote monitor acquisition as an extension of our remote method invocation mechanism. When a remote monitor is acquired or released we send a message to the appropriate node, specifying the object to be operated upon. This message is handled similarly to a remote method invocation; if a representative Java thread does not yet exist for the global thread we create it. Otherwise, we know that the thread is waiting inside the `waitForMessages` method. Thus, we can add a mechanism to `waitForMessages` that obtains and releases monitors for the global thread.

The design of this mechanism is complicated slightly by the bytecode-level semantics of Java monitors [Lindholm and Yellin, 1999]. Each `MonitorEnter`/`MonitorExit` pair must be balanced within their method's body; every control flow path through the method must have exactly one monitor exit operation for every monitor entry. This means that we cannot simply generate a method that acquires the monitor for an object. Such a method would have to release the monitor before exiting, which would violate its purpose, or would generate a run-time error. Instead we nest remote monitor entries and exits within the remote method infrastructure, as shown in Listing 4.4.

We extend `waitForMessages` by handling two new message types: an `EnterMonitorMessage` at line 10 and an `ExitMonitorMessage` at line 13. When a remote node enters a monitor, it calls `enterMonitor`. First it finds the target monitor object (line 20). For the sake of clarity, we assume that the target is local to the node. This is normally the case, since the requesting node must necessarily have obtained a reference to the object in order to acquire its monitor, and so will also have obtained location information. We handle the corner case in which the requesting node's information is outdated (due to migration of the target) through the `RemoteObjectManager` service.

Given the appropriate local object, line 21 ensures that we hold the canonical object (rather than a wrapper, as discussed earlier). We then acquire the object's monitor with a **`synchronized`** block, and make a nested call to `waitForMessages`. Execution then proceeds as before, with remote method invocations forming a stack of nested calls. Even-

```
1  public Object waitForMessages() {
2    for(;;) {
3      Message msg = Message.recv(thread_id);
4      if (msg instanceof MethodResultMessage) {
5        ...
6      }
7      if (msg instanceof InvokeMethodMessage) {
8        ...
9      }
10     if (msg instanceof EnterMonitorMessage) {
11       enterMonitor((EnterMonitorMessage)msg);
12     }
13     if (msg instanceof ExitMonitorMessage) {
14       return null;
15     }
16   }
17 }
18 private void enterMonitor(EnterMonitorMessage msg) {
19   Object monitor
20     = RemoteObjectManager.findObject(monitor.getUID());
21   monitor = MonitorManager.getLockableObject(monitor);
22   synchronized(monitor){
23     waitForMessages();
24   }
25 }
```

Listing 4.4: Remote monitor acquisition

tually the remote method will encounter a monitor exit operation (guaranteed by bytecode semantics), at which point it will send an `ExitMonitorMessage`, returning from the `waitForMessages` invocation at line 23.

4.3.5 Exception Handling

When executing a remote method in RuggedJ we must preserve any exceptions thrown by that method. An exception thrown by a remote method may be caught by a local **`catch`** clause, and so to preserve the semantics of the original application we must ensure that such an exception is thrown.

Java's exception handling mechanism limits our rewriting capabilities for exceptions. An object thrown or caught as an exception or error must inherit from Java's `Throwable` class. This rules out the Wrapping rewriting template, since Wrapped types do not preserve the inheritance hierarchy of the original type. Recall that Wrapping is the mechanism by which we ensure that all system classes can conform to the RuggedJ object model; a `Throwable` subclass in the Java standard libraries which could not be rewritten using any other template could therefore fail to conform to our object model. No such class exists within any of the standard libraries that we have examined. In practice, all `Throwable` subclasses that we have examined are classified as Direct.

We treat the exceptional exit from a method in a similar manner to a standard return. All remote method invocations are performed as reflective calls on the executing side. Under Java's reflective semantics, any exception that remains uncaught in reflectively called code is wrapped in a `InvocationTargetException` and thrown to the invoking code. We wrap our reflective method invocations in a **`try`**/**`catch`** block that catches this exception. We unwrap the original thrown exception and pass it back as part of the response to the original method invocation message. On the invoking node we check all method message responses for exceptions, re-throwing any that have been raised. This way the original exception is thrown and can be caught by any calling context.

The re-throwing mechanism contains an interesting subtlety. Thrown exceptions must be declared in method signatures to statically ensure that calling code can handle any such exception. Thus, our remote invocation method must declare that it throws `Exception` (as it can throw any subclass of `Exception`). Normally, this would require that every possible control flow path to this method contain a **`catch`** block that can handle `Exception`. This

is not an issue within RuggedJ, since the remote method invocation code is called only by stub object, which are generated at the bytecode level. As these classes are never compiled, the requirement that thrown exceptions be catchable is not checked.

One unfortunate limitation of our exception handling mechanism is that stack trace information is not correctly preserved. The stack trace for an exception starts at the point where it was thrown on the local machine (inside RuggedJ's messaging system), losing any trace from the remote execution. We could solve this in several ways. First, rather than re-throwing the original `Throwable` object we could create a new object of the same type, and chain the original exception object as a "cause". This would preserve the stack trace information, albeit with some additional frames for RuggedJ's remote invocation infrastructure. However such an approach would risk losing additional information stored in the exception object, and so could alter the semantics of the original application. Alternately, we could insert correct stack trace information into the `Throwable` object before it is re-thrown, composing the current stack trace with that on the remote machine. This would give accurate stack trace information, but would carry the overhead of generating unnecessary stack traces for all remote exception whose traces are not examined.

4.4 Application Partitioning

In order to distribute an application across a RuggedJ network, we must determine which objects are to be allocated upon which nodes. We refer to this process as *partitioning* the application. RuggedJ provides a partitioning interface to which developers can write a partitioning policy. We believe that the application developer is in the best position to provide an optimal partitioning, guided by the output of our whole-program static analysis.

While creating a partitioning policy for an application may seem to be a daunting task, we have found that simple policies generally perform well; the policies for each of the benchmarks discussed in Section 5.3 never required more than a few dozen lines. As a general strategy, we first locate the root object of a distribution unit (typically a `Runnable` object that contains the work for an individual thread), and allocate instances of this ob-

```
1  public class DatabasePartitioning extends Partitioning {
2    private int allocated = 0;
3    protected int loadTimeAllocationPolicy(AllocationSite site) {
4      if (site.getTargetClass().equals("search.ComparisonThread"))
5        return ALLOCATE_DYNAMICALLY;
6      return ALLOCATE_LOCAL;
7    }
8    protected int runTimeAllocationPolicy(AllocationSite site) {
9      if (++allocated > NodeManager.getMyNode().availableProcessors())
10       return ALLOCATE_REMOTE;
11     return ALLOCATE_LOCAL;
12   }
13   protected Node runTimeAllocationNode(AllocationSite site) {
14     int id = 1 + (allocated % (NodeManager.getNodeCount() - 1));
15     return NodeManager.getNodeByID(id);
16   }
17   protected List<RJType> getDeclaredReplicable() {
18     List<RJType> replicable = new ArrayList<RJType>();
19     replicable.add(RJType.get("char[]"));
20     return replicable;
21   }
22   protected boolean copyTransitiveClosure(Object obj, RJType t) {
23     return t.equals(RJType.get("search.Chunk")) ||
24          t.equals(RJType.get("search.ResultSet"))
25   }
26   protected List<String> getNonRewritable() {
27     List<String> local = new ArrayList<String>();
28     local.add("neobio.alignment.*");
29     return local;
30   }
31 }
```

Listing 4.5: Partitioning policy for DNA database matching application

ject on remote nodes. By default, any subsequent allocations are performed locally, so objects related to that distribution unit are automatically placed on the same node. Listing 4.5 shows the partitioning policy for one of our benchmarks, a DNA database matching application, excluding those parts related to migration.

Remote allocation is decided in stages, to afford maximum flexibility. During the bytecode rewriting phase the partitioning policy is expressed for each static allocation site using method `loadTimeAllocationPolicy`. Using the supplied allocation-site information, the policy can determine whether objects allocated at that site should be allocated locally, remotely, or dynamically. In the case of dynamic allocation, the policy is queried again at run-time, whenever that allocation site is executed, as expressed by method `runTimeAllocationPolicy`. At that point, the policy can determine whether to allocate locally or remotely, depending on the dynamic condition of the network. Remote allocations, whether determined statically or dynamically, cause the partitioning policy to be queried an additional time to determine the node at which the remote object should be allocated, as specified by method `runTimeAllocationNode`. The policy has access to the run-time library's network metadata, including the number of nodes and the capabilities of hosts on the network, so an informed decision can be made at run-time. Here the policy cycles between all nodes other than the head node (number 1).

The partitioning policy also allows a developer to declare types to be immutable, allowing them to be replicated. One of the uses of this mechanism is when the instances of a class undergo a population phase, after which they are not modified. Our static analysis cannot detect that the object is not modified after a certain point in the application, and so marks the class as mutable. However, based upon knowledge of this phase behavior, the developer can declare the class to be immutable for the purposes of distribution, allowing it to be replicated across the network, using the `getDeclaredReplicable` method. The database matching application operates over large character arrays that are initialized early in the application, and then used unmodified. Line 19 allows these arrays to be replicated. Note that this occurs at the type level, and so requires that all **`char`**`[]` arrays in the application are immutable.

The `copyTransitiveClosure` method allows the developer to designate classes that should be copied using Java's serialization mechanism rather than RuggedJ's reflective copying technique. This offers several key advantages: serialization is much faster than RuggedJ's marshaling and unmarshaling process, and it allows multiple objects to be copied in a single operation. However, the classes that can be copied in this way are limited. Clearly, such classes must implement the `Serializable` interface. Additionally, all classes in an instance's transitive closure must be direct (i.e., they are not rewritten to implement the RuggedJ object model). Serializing instances of a class `X` that does implement the RuggedJ object model would copy `X_local` instances, which violates the rule that every local instance represents a unique object.

The final method in Listing 4.5, `getNonRewritable`, allows the developer to specify classes that will not be rewritten. We treat these objects as system classes, and load them into the JVM without modification. This mechanism exists as an optimization to allow key performance-critical sections of code to execute without the overhead of interface indirection, such as accessing public members and array elements via get and set methods. These non-rewritable classes may not reference any rewritten classes, as they will be unaware of the RuggedJ object model. Additionally, they may not reference non-final static data, since they cannot access static singletons.

The partitioning policy is also responsible for managing object migration. Listing 4.6 shows the migration policies for the DNA database application. The partitioning policy must first declare which classes may have migrating instances (`mayMigrate`). This allows us to allocate proxies only for those classes that may need them, avoiding the indirection overhead for all other accesses. Migration can be triggered in three ways. The first is to migrate the return value back to the caller (`migrateMethodReturnValue`). In this case, the return value of a method called from another node is immediately migrated to that node, allowing it to be operated upon locally.

The second trigger is after a remote object has been invoked some number of times up to a fixed threshold (`migrateFrequentlyCalled`). This allows objects to migrate to

```
protected boolean mayMigrate(RJType t) {
  return t.equals(RJType.get("search.Sequence")));
}
protected boolean migrateMethodReturnValue(UID uid,
    String methodname, String methoddesc, Object[] args) {
  return false;
}
protected boolean migrateFrequentlyCalled(Object obj,
    RJType type, UID uid, int callcount) {
  return false;
}
protected List<MigrationTrigger> getMigrationTriggers() {
  List<MigrationTrigger> triggers
    = new ArrayList<MigrationTrigger>();
  triggers.add(new MigrationTrigger("search.Sequence",
    "populate", "(Ljava/io/BufferedReader)V", Position.End,
    "this"));
  return triggers;
}
protected Node
migrationTriggerCallback(MigrationTrigger trigger, UID uid)
{
  int id
    = 1 + (allocated % (NodeManager.getNodeCount() - 1));
  return NodeManager.getNodeByID(id);
}
```

Listing 4.6: Migration policy for DNA database matching application

whichever node is currently accessing it most frequently (once the threshold is reached), letting the partitioning strategy account for phase behavior.

Finally, a partitioning policy can install callbacks at the start or end of arbitrary methods in the application (`getMigrationTriggers`). The policy specifies the class, method

name and signature into which a callback is to be inserted, as well as the position (start or end) within the method. Additionally, it gives a variable name that may be migrated. A callback to the partitioning policy is inserted in the specified method when the bytecode is rewritten, so the policy is called when the method executes at run-time. The callback method (`migrationTriggerCallback`) allows the partitioning policy to supply a node to which the referenced object should migrate. This mechanism allows arbitrary values (such as local fields or method arguments) to be migrated, rather than simply the return values of methods.

4.5 Contributions

The primitives upon which we build our run-time infrastructure have some similarities to those implemented in other systems. Redirecting remote method invocations through stub objects is a common theme in object-oriented distribution, although our implementation does not rely on RMI as does several other projects. By implementing our own communication layer we are afforded more flexibility in our run-time, allowing us to implement thread affinity. RuggedJ's communication system also allows different forms of message passing for different goals; broadcast messages use a tree-based communication hierarchy, while method invocations go from node to node. This dual-mode messaging system significantly reduces communication overhead.

RuggedJ's network configuration is also a key difference with previous work. The majority of transparent distribution systems (J-Orchestra, Addistant and AIDE) focus on small networks with fixed computing resources. The only other system that targets larger clusters of machines is Terracotta. However, Terracotta uses a similar distribution model to the simpler systems, with a client/server architecture that offloads work from a central server to worker nodes. RuggedJ deemphasizes the head node (necessary to coordinate a few global activities), while storing canonical versions of objects across the network and implementing a peer-to-peer configuration.

The idea of object migration for spatial locality is also a common theme within distributed systems, going back to an early implementation in Emerald. However RuggedJ's unique partitioning plug-in interface allows developers to combine application knowledge with a level of dynamic introspection that is not commonly available within transparent distribution systems. In addition, some other distribution systems (including Emerald and Addistant) allow immutable objects to be replicated on multiple nodes. However they define this immutability at the object level, whereas we detect determine immutable fields which we replicate using caching in our stubs.

5 DISTRIBUTED APPLICATION DEVELOPMENT

The bytecode transformations and run-time infrastructure that we have presented in the previous chapters allow us to support the vast majority of Java applications. However, not all applications benefit from distribution. We target a class of applications that we refer to as *distributable*: applications that can be broken into multiple discrete units, each of which is operated upon by a dedicated node. In this chapter we will describe some of the design decisions that can impact the distributability of an application, based upon our experiences while writing and modifying applications to run on RuggedJ. We will then discuss some concrete examples, detailing how we modified and partitioned the applications to maximize performance under RuggedJ.

Many of the application characteristics that we discuss in this chapter are applicable to distributed systems in general, while others exist as artifacts of RuggedJ's distribution mechanisms. We therefore approach this issue in two parts. Section 5.1.1 outlines the concept of distributability in Java and discusses some of the fundamental properties that distributable applications must demonstrate. Section 5.1.2 talks more specifically about targeting applications to the RuggedJ infrastructure, and how design decisions in the application can make the most of our distribution platform. Section 5.2 describes the general strategies that we use when partitioning applications. Finally in Section 5.3 we describe some large, realistic applications that we have either designed or modified to distribute with RuggedJ.

5.1 Distributability

The constant CPU performance improvements implied by Moore's law [Moore, 1965] have meant that developers have traditionally been content to scale their applications *vertically*. Vertical scaling is achieved by adding more and faster hardware to a single machine;

improvements in the platform lead to better performance at no cost to the developer. Such a scaling model is clearly attractive. We can simply wait for the next generation of hardware to improve our applications' performance. Modern hardware trends, however, do not offer the same vertical scaling opportunities. Clock speeds have reached a plateau, with additional computing capacity coming from additional cores rather than from increases in monolithic processor horse-power.

This shift in hardware design must be met by a corresponding change on the software side. Rather than scaling vertically, applications must be designed to scale *horizontally*, finding performance gains by executing across multiple cores and machines. Developing such applications requires a new way of thinking for those used to vertical scaling; not only must applications be multi-threaded, but data structures and access patterns must be designed to produce discrete work units that can be operated upon independently of one another. Decomposing applications in this way requires more effort at the design phase, and may require more complex synchronization than a simple single-threaded system. To compensate, however, horizontal scaling allows a well-designed application to scale out to a virtually limitless degree.

5.1.1 General Distributability

In this section we discuss some of the design decisions that allow an application to scale horizontally. We focus here on general principles that hold true regardless of the distribution mechanism used. We will discuss RuggedJ-specific techniques in Section 5.1.2.

Application Structure

Distributable applications must be structured with natural distribution points, with each *distribution unit* performing a subset of the application's workload. Scaling can thus be achieved by executing more distribution units on additional hardware resources, rather than by increasing the workload for a single unit. This stands in contrast to the traditional single-threaded application model, where vertical scaling allows a single unit to perform

more work in a single machine. Each distribution unit should be operated upon by one or more dedicated threads. By binding threads to a specific part of the application's data and workload we can partition distribution units among the cores or machines of a system while preserving spatial locality in a given thread's accesses.

The application should be structured with minimal interaction between distribution units. When scaling to a large number of units we may not be able to predict the latencies between any two; the data from a remote unit may be located within the shared memory of the local machine, in a distant memory bank in a NUMA architecture, or on a remote machine. Thus, it is likely that any cross-talk between distribution units in a large application may translate to expensive network operations.

Finally, the design of data structures within a distributed application is crucial. Data must be allocated close to the thread that will use it in order to avoid costly remote accesses or migrations. Thus data structures that can be decomposed into smaller units are preferable to single indivisible data structures. For example, large arrays pose a barrier to distribution; such a structure must be allocated on a single node, whereas a list of smaller arrays could be spread across the network.

Distribution Bottlenecks

We have encountered several common bottlenecks that keep applications from scaling horizontally. The most common is mutable static state (as opposed to immutable static data such as class constants which can be replicated, and so does not pose an issue to distribution). An application's mutable static data must be globally consistent across the system. This means that the state must either be stored on a single node, with all updates and reads being performed on that node, or it must be subject to some form of coherence mechanism to ensure that updates on one machine are reflected on all others. RuggedJ uses the former mechanism. Either one of these approaches leads to an increase in network traffic and can slow down data accesses, causing a deterioration in system performance. Application designers can alleviate this overhead by minimizing the use of mutable static data,

using shared locally-consistent data rather than globally-consistent statics where possible. A similar restriction affects the usage of the Singleton design pattern [Gamma et al., 1995]. By encapsulating a class's state in a single object, developers risk such an object becoming a bottleneck for distribution.

Similarly, data sources that conform to the data access objects (DAO) pattern can present difficulties in a distributed system. Forcing all data accesses to go through a single DAO instance requires that all data is accessed from a single machine. In the same vein, several construction design patterns can lead to bottlenecks. Builders and factories that maintain internal state must result in remote calls to construct objects. Such construction patterns could also produce extra migration overhead, as the newly-created object must then be relocated to the node upon which it is to be used.

Immutability

Immutable data offers the largest performance improvement in a distributed application of any factor beyond basic application structure. Since the value of immutable data will never change after an initial setup phase we can replicate it on each node in the network, allowing local read access to all threads. Application developers can increase the immutable content of their applications by factoring out the mutable content of a class. This allows the immutable content to be stored locally, while the mutable content is subject to remote invocations. Note that the RuggedJ run-time system makes this optimization unnecessary by identifying and caching immutable state in the stub; performing such a transformation by hand could be error-prone in case of later modifications to the application.

The design of class initialization can also increase the immutable content of a class. By setting immutable fields in a class's constructor such fields can be marked as **`final`**, and so can be picked up by analysis tools. Contrast this to setting the fields in an initializing method that is called after the object is created. While the fields are still immutable, it is not as clearly obvious without a more involved control flow analysis.

Object Migration

A factor to consider when designing data structures is object mobility. Migration allows us to exploit shifting access patterns and maximize spatial locality. However, the act of migrating itself may cause unacceptable overhead. Should two or more threads on remote machines refer to a single object, there is a risk that the object may "ping-pong" between the machines, migrating from one to the other depending on the source of the most recent access. While this can be alleviated in part by a sensible migration policy, a better approach would be to avoid this behavior in the application's design.

One approach to limiting spurious migration is to define phases within the application or object's life cycle. The object is then bound to a particular distribution unit for the duration of the phase, preventing it from migrating before the next phase boundary. Note that this does not affect the reachability of the object; it can still be accessed by remote machines, just not migrated. An example of this would be the creation and use of a large data structure. The structure would be bound to the creating unit until it has been populated, and then to a different unit that makes use of it after it is complete.

Serial Sections

An issue that can seriously limit the distributability of an application is sections of serial code. The maximum performance gain of a distributed system is limited by the percentage of the application that is parallel [Amdahl, 1967]. Thus, by introducing serial sections of code we limit the benefit that we can see from horizontal scaling.

Some serial code is unavoidable. It is frequently necessary for a single controller thread to handle global tasks such as creating distribution units, partitioning work between them, merging results, and so forth. However, there are some cases in which serial execution is unnecessary. Threads in Java are implemented as classes implementing the `Runnable` interface; calling the `start` method on a thread associated with such an object will cause the `Runnable` class's `run` method to execute in a new thread. Only code invoked from the `run` method is executed in the new thread; the object's constructor is executed by the parent.

We have seen cases where a series of worker threads are created by a single controller, with substantial work performed in each `Runnable` object's constructor. Moving this work to the start of the `run` method has increased the distributability of the application, and brought considerable performance improvements.

5.1.2 Designing for RuggedJ

RuggedJ offers significant advantages to developers of distributable applications, including a shared-memory abstraction, flexibility in network infrastructure, caching of immutables and straightforward object migration. If a developer plans to leverage the RuggedJ infrastructure, there are some additional design choices that can be made to maximize the benefit from the infrastructure.

Language Features

There are several features of the Java language that we do not support within applications, and others for which we provide limited support.

We do not support applications that define their own user-level class loaders. This is the case for two reasons. We rely on the fact that we transform all user classes to implement our object model. Allowing applications to install their own class loaders could violate this constraint, allowing unmodified classes to exist in the system (for similar reasons, we do not allow applications that use the JVM Tool Interface). The second cause to disallowing custom class loaders is that we use a full-program analysis to determine which template should be used to transform each class and to identify immutable data. The result of this classification depends on all the classes in the system, with the result for one affecting the classification of others. Generating unprocessed classes at run-time through a custom class loader may invalidate the results of this classification, and can lead to mutable data being replicated.

We also do not support Java's security policies. RuggedJ is implemented entirely at the user level, but performs some minor modifications to system classes using the JVMTI.

To avoid security exceptions, we have defined our own, highly liberal, security policy. Application-level security policies may be more strict than our own, and so can prevent RuggedJ from functioning properly.

We offer limited support for reflection within RuggedJ. As we have described in Section 4.3.2 we use a heuristic-based system that intercepts reflective calls and rewrites them to work within our rewritten infrastructure. However we do not guarantee that these transformations preserve the behavior of the original application. Thus, while many uses of reflection will work as intended in our system, we cannot fully support all reflective semantics.

Simplifying the Object Model

The RuggedJ object model as described in Section 3.2 allows all system classes to operate within a RuggedJ network. It does this using a range of implementation techniques that work around the limitation that we cannot directly transform library code. Since we can rewrite user code, we have far more freedom in transforming such classes. However there are two instances in which we must apply the Wrapping or Extending template to user classes.

First, as we discussed in Section 3.5, a user class must use the Wrapping template if its superclass is classified as Wrapping. This can come about if the user class extends a Wrapping system class; the user classes and any subclasses will be classified as Wrapping, and will incur extra wrapping and unwrapping overhead. It is difficult to predict ahead of time how a given class will be classified (classification for each class depends on the classification of other classes in the closure; adding new system classes can change the classification of existing classes). Thus it is advisable to minimize subclassing of system classes.

The other way in which a Wrapping or Extending template may be applied to user code is through exposure to native code. As we have previously stated, native code can break our transformed applications, should a rewritten class or member be referenced. To minimize

this risk, if our heuristics should suggest that a class may be exposed to native code we do not rename it, and so must use the Wrapping or Extending template. Additionally, exposure to native code limits code mobility, as an object that may be referred to by native code cannot migrate. We therefore recommend that applications targeting RuggedJ do not include native code.

Replication

RuggedJ offers more flexibility in identifying immutable code than is available through the Java language alone. First, we use our whole-program analysis to identify immutable data, rather than relying on the **`final`** keyword. More importantly, however, we allow developers to identify functionally-immutable data through the partitioning policy. This takes advantage of the developer's application-specific knowledge; a class may be statically determined to be mutable, but may have its contents frozen after an initial setup phase, allowing it to be replicated safely. This maximizes the data that we can replicate in a distributed application. Developers can take advantage of this fact when designing their objects' life cycles.

As a further optimization, developers can declare immutable classes to implement the `Serializable` interface where practical. This way we can take advantage of the faster data transfer gained by serializing objects, as discussed in Section 4.2.2.

Direct Field Access

Contrary to most Object-Oriented development best practice advice, fields in RuggedJ should be accessed directly, rather than using `get` and `set` methods. The reason for this is subtle: the RuggedJ class loader rewrites all `GetField` bytecodes to be method invocations on generated interfaces (with a symmetrical implementation for `SetField` bytecodes). The `get` method is implemented differently depending on the object's location; a `get` method in a local class simply returns the field, while the corresponding method in a stub performs a remote method invocation. However, we optimize stub methods in the case of immutable

fields. Should a field be determined to be immutable we cache the value in the stub, and so save the overhead of a remote method invocation. Thus, `GetField` bytecodes on immutable fields are purely local operations, regardless of the object location.

In the contrary case, a user-defined `get` method in the original application is not accessed using the `GetField` bytecode, but is called using `InvokeVirtual`. It is therefore not redirected to our generated `get` method. In the local case, the user-defined `get` method performs in the same way that the generated version would. However, the stub implementation of such a method simply performs a remote method invocation on the local object to execute the method. Thus, even if a field is immutable calling a `get` method in the original application can lead to a remote invocation.

Performance-Critical Sections

While the RuggedJ object model allows all data to be remotely referenced, it incurs some performance overhead. Redirecting all field accesses through `get` and `set` methods carries a performance penalty, particularly when iterating through the elements of an array. RuggedJ allows developers to avoid this run-time overhead by designating performance-critical classes to be System Direct. This means that such classes are loaded into the VM with minimal transformations, and so run at the same speed as they would without RuggedJ.

We can perform the System Direct modification on any classes that are known to be immutable or purely local. Any remote reference to a System Direct class leads to its replication, which would cause lost updates in the case of a mutable object. This requirement is not particularly arduous — the performance-critical sections of well-partitioned applications should be purely local (otherwise such sections would require frequent remote invocations, making the indirection overhead insignificant).

Of more concern is the fact that System Direct code cannot refer to transformed types. The transformations that would make System Direct code refer to new types (indirection through interfaces, calling `get` and `set` methods in case of remote data, etc.) are exactly the performance-affecting transformations that we seek to avoid. Thus, we can use this

optimization only on those performance-critical classes that do not refer to shared objects. By structuring their applications to take this consideration into account, developers can maximize the classes that they designate System Direct, and so minimize the performance impact of the RuggedJ transformations.

5.2 Partitioning Strategies

We have generally found simple partitioning strategies to work well. In each of the applications that we discuss in Section 5.3 we have determined a clear point for decomposition into distribution units. We found that distribution units are normally anchored in a `Runnable` object (i.e., the root of the distribution unit is bound to a single thread).

Once we have identified the distribution units, we must determine the nodes upon which to place each unit. This will generally depend on the resources available to each host; we want each distribution unit to have a roughly equal level of computing resources. This way we do not have one unit lagging behind the others, leading to lower overall performance. Our partitioning strategies begin by introspecting into the nodes of the network, determining the number of cores available to each. We aim to assign at most one distribution unit to each core to avoid thread switching (naturally this is possible only until the cores of all nodes are saturated). The allocation of distribution units to nodes depends on the number of units in the system, as well as the capacity of individual nodes. We consider three load levels:

Light load. A system is lightly loaded when the number of distribution units is less than or equal to the number of cores available to the head node. In this case we allocate all distribution units to the head node. The head node is the natural place to allocate the first distribution units; we know that the application is launched on the head node, and standard input and output streams are redirected to the head node's console. By allocating the maximum number of distribution units on the head node we eliminate remote references between those units. The only exception to this is where there is

considerable work performed by a controller thread that is tied to the head node. In this case the head node may be saturated before we allocate any distribution units.

Medium load. A medium load is when the number of distribution units is marginally greater than the number of cores available to the head node. The definition of "marginal" in this case depends on the level of inter-node connectivity. Under a medium load we allocate units on the head node until it is close to capacity, and place the remaining distribution units on the second node. We do not completely saturate either node since there will be incoming remote method invocations that must be handled; allocating units to every core of a node would lead to thread switching should an incoming method request arrive. The more cross-talk that exists between distribution units, the more cores should be left available to fulfill remote requests.

Heavy load. A heavily loaded system is one where the number of distribution units is far greater than the number of cores available to the head node. In this case we allocate the distribution units evenly across the nodes. This allows maximum flexibility for handling interactions between distribution units. If such interactions are few, there is little penalty for allocating units on remote nodes. If interactions are more common this approach spreads the targets for remote method invocations across the network, and leaves the maximum number of cores available to handle incoming requests.

These strategies are general guidelines, rather than hard-and-fast rules. They make assumptions about the application (such as a uniform pattern of access to remote data) that may not hold true for any given circumstance. For example, some of the applications that we studied had a central controller object to which results are returned after each distribution unit completed its work. In this case it was advantageous to saturate the head node (to avoid copying results from one node to another) even when the system was under a medium or heavy load. This is one of the major benefits of our partitioning system; an application developer can easily specify a customized allocation policy for an individual application.

5.3 Applications

Clearly, not all applications meet the distributability requirements laid out in the previous section. Many computational tasks have data dependencies that make them inherently serial, or limit their scalability to a finite number of threads. Others would appear to be distributable, but have serial sections that overwhelm any speedup gained by distribution. However, we have identified a number of classes of applications that demonstrate the qualities that we rely upon for distribution:

Scientific computation. Scientific calculations such as physics simulation, financial analytics, genetic computation or fluid dynamics lend themselves well to distribution. Such problem sets generally have a small amount of data upon which expensive computation must be performed. These calculations can often be performed in parallel, taking advantage of extra processing capacity.

Rendering. Rendering a three-dimensional scene by ray-tracing is a computationally intensive process in which individual rays of light are simulated interacting with objects. Rays are independent of one another, and so can be traced in separate distribution units.

Business software. There are many accounting and middleware applications that could distribute under our system. For example, tax accounting software requires many independent calculations based on individual transactions. These calculations could be performed in parallel. Additionally, financial trading firms make use of complex models to simulate markets; such scenarios can be simulated in parallel. Finally, many businesses produce complex reports that must be generated from large data sets that can be decomposed to run on separate machines.

Functional-style programming. A final class of applications are those written in a functional style, with few side-effects to methods. An important example of this programming style is map/reduce, implemented by Google's internal infrastructure [Dean and Ghemawat, 2008] or by the open-source Hadoop framework [The Apache Software

Foundation]. This programming paradigm is ideally suited to distribution, at it makes explicit the requirement that work units be independent.

In this section, we will examine several applications from diverse problem domains. We will discuss how each was designed or modified to distribute under RuggedJ, demonstrating the principles outlined in Section 5.1. We will outline the partitioning and migration policies for each application and, where appropriate, present the scalability characteristics when each application is run on a RuggedJ network. Our results were gathered on a small cluster of three machines. Each 16-way host had eight dual-core AMD Opteron processors, running at 4.5GHz, and 32Gb of RAM, in a NUMA arrangement. The machines were connected on a private Gigabit Ethernet switch. We measured both the untransformed application running on a single host, and then the RuggedJ transformed version as it runs on multiple hosts. We present results showing 95% confidence intervals, gathered from 30 iterations for each data point. We normalize scalability curves to the untransformed wall-clock run time of the application using four threads; normalizing to the execution time with a single thread would produce a similar curve, but gathering single-core execution times for the large applications that we run over multiple iterations would have taken an unreasonable length of time (on the order of weeks per data point).

Our benchmark applications come from a number of sources:

Java Grande. We studied two of the large-scale applications from the Java Grande multi-threaded benchmark suite [The Java Grande Forum]: MolDyn, a molecular dynamics n-body simulation, and MonteCarlo, a financial simulation using Monte Carlo pricing techniques. RayTracer, the third application, was unsuitable because its memory access patterns on our NUMA architecture lead to such large variations in execution time that performance cannot be measured consistently.

DNA Database Matching. This application was adapted from the DSEARCH application by the Heterogeneous Distributed Computing group within the Department of Computer Science at the National University of Ireland Maynooth [Keane and Naughton, 2005]. It compares a set of protein sequences against a database to identify similari-

ties. The original application ran under an explicit distribution harness, similar to the BOINC infrastructure [Anderson, 2004].

SPECjbb2005. We experimented with SPECjbb2005, a standard Java benchmark that simulates transaction processing within a business database. However, we found that, contrary to our expectations, SPECjbb2005 was not distributable. Instead, we developed a new application based upon SPECjbb2005 that performed the same workload in a distributable manner.

Clue. We developed a multiplayer version of the board game *Clue* that we distributed with RuggedJ. While performance was not an issue in this application (the time spent by a player making a move far outweighed any communication overhead) this application demonstrates the ease with which complex network protocols can be encapsulated by RuggedJ, as well as our capability to distribute applications that rely on complex Java libraries such as Swing [Robinson and Vorobiev, 2003].

5.3.1 Monte Carlo Simulation

Our first benchmark from the Java Grande suite uses a Monte Carlo simulation to derive the price of a product from the price of an underlying asset. This benchmarks is an example of the many real-world financial simulation problems that can benefit from distribution.

Application Overview

The Java Grande project provides two versions of each benchmark application: a multi-threaded shared memory version and a message passing version that uses MPJ. Neither version of the Monte Carlo benchmark was ideal for transparent distribution. The shared memory implementation made heavy use of static arrays, causing a bottleneck as outlined in Section 5.1. Removing this shared data was required to make distribution practical; a straightforward distribution of the shared memory version was possible, but took an unfeasible amount of time to run due to repeated remote invocations. The MPJ implementation,

on the other hand, used localized data structures but built distribution around explicit remote calls. We therefore created a hybrid version of the benchmark that used the control logic of the shared memory version with the non-static data structures of the MPJ version. This version eliminated the complexity of explicit MPI invocations, while allowing the application's data to be split into discrete units.

The Monte Carlo benchmark code also contained significant debugging code which was controlled by a **`static boolean`** value in each class, as well as a per-class **`static`** `String` value that was prepended to any debug messages. These values were set during an initialization phase to a **`false`** value hard-coded into a controller class. While inspection of the code told us that each debug flag could never be set to true, the fields could not be cached, as they were not final, and could not be overridden in the partitioning policy (at present, partitioning policies can specify immutability at the class, not the field level). Thus, every piece of debug code, while never executed, triggered a call to the static singleton. To solve this problem, we set the per-class prompt field to be **`static final`**, using the value assigned in the initialization phase. This way we could cache the field in the static stub. We also factored the per-class **`static boolean`** debug flag fields out to be a single **`static final`** field in the controller class (from where the hard-coded **`false`** value was propagated), allowing it to be referenced from all other classes and cached in the static stub.

Finally, we modified the benchmark to make it perform more iterations of the workload. The Java Grande suite was released in 1999 and reflects significant workloads for the time. However, what was a large workload in 1999 can be trivially executed by today's hardware. We introduced a loop that simply calculated the price for each stock five hundred times; this way the application ran for approximately one hour using four threads on one of our test machines.

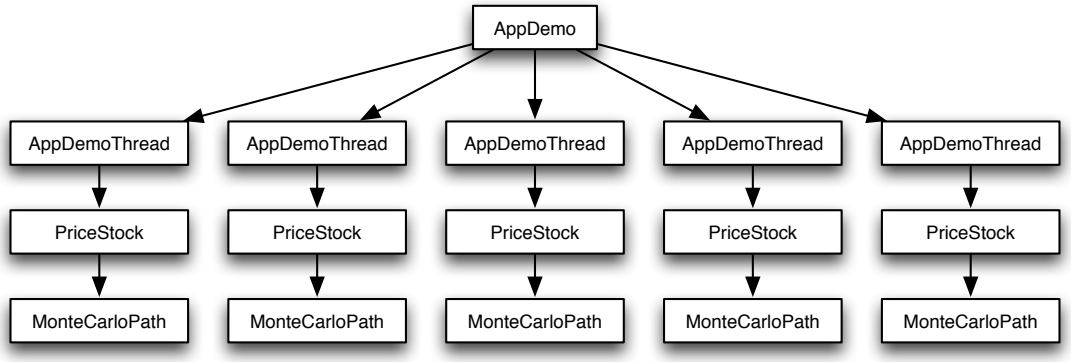


Figure 5.1.: Monte Carlo application structure

Partitioning

The core of the Monte Carlo benchmark is structured as shown in Figure 5.1. A run of the application is initiated in the `AppDemo` class, which creates multiple `AppDemoThread` objects. Each `AppDemoThread` is executed in a separate thread, and so form the natural units for our decomposition. Each of the `AppDemoThread` objects processes a part of the workload, creating local `PriceStock` objects that perform the Monte Carlo simulation. Once all stocks have been priced, the `AppDemoThread`s pass the result back to the `AppDemo` object.

We define a distribution unit to be a single `AppDemoThread` object. Any temporary data (such as `PriceStock` or `MonteCarloPath` objects) is allocated locally and so is collocated with the parent `AppDemoThread` object.

The shared data within the Monte Carlo benchmark consists of a large array of `ToTask` objects, each of which represents a stock to be priced. We declare the array type `ToTask[]`, as well as the base type `ToTask` to be immutable, allowing us to replicate this array. We can do this because neither the contents of the array nor the fields of each `ToTask` object change after an initial setup phase. Each `AppDemoThread` operates over a section of this array, defined by a thread identifier passed when it is created. While this results in some unnecessary replication (not all replicated `ToTask` objects are operated upon by each node),

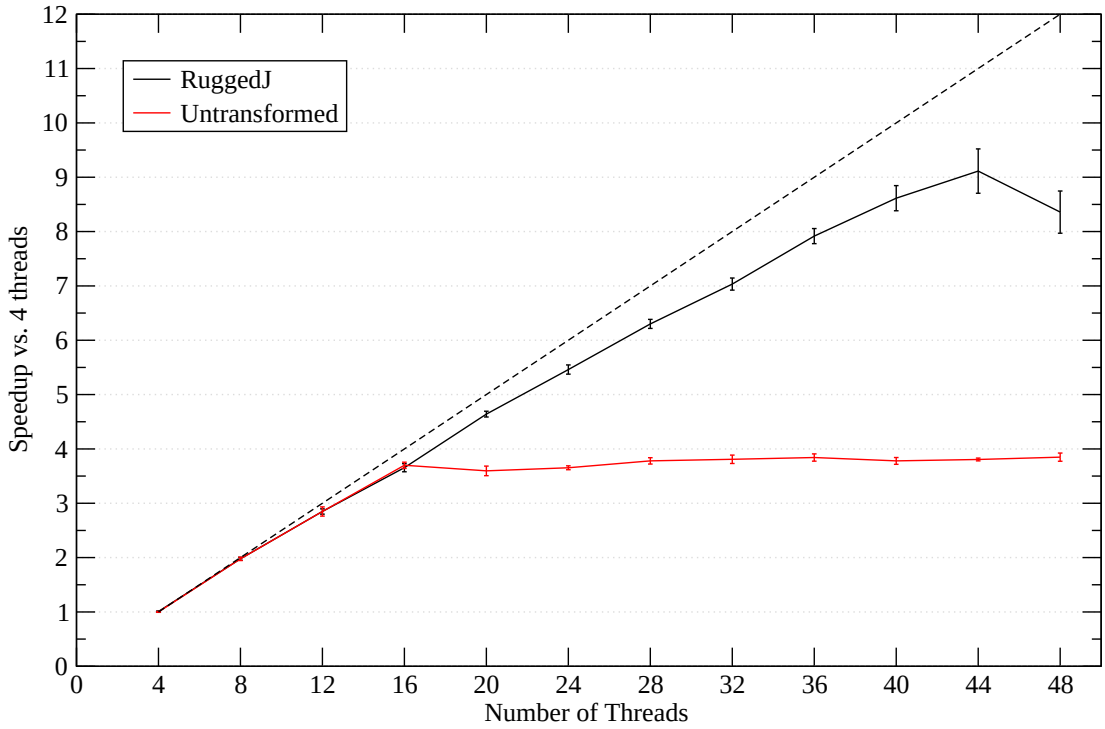


Figure 5.2.: Monte Carlo speedup (normalized to untransformed)

we use serialization to make the replicas, resulting in negligible overhead. Finally, each thread passes the result of its computation back to the main `AppDemo` object, in the form of an array of `ToResult` objects. We treat the `ToResult` array in the same way as the `ToTask` array, replicating the results on the original node.

We declare the majority of classes within the benchmark to be System Direct for performance reasons. The only mutable remotely referenced classes in the system were the main `AppDemo` class and the per-thread `AppDemoThread`. These were User Unconstrained, while the remainder of classes were declared to be System Direct.

Performance Evaluation

Figure 5.2 shows speedup for our rewritten Java Grande Monte Carlo simulator, running in both its untransformed form and under RuggedJ. We see that the untransformed version of the application experiences performance improvements as we add threads until it reaches

16 threads, after which it flattens out. This is to be expected, as each host has 16 cores. In contrast, RuggedJ shows steady scaling as it utilizes the second and third nodes in the cluster. For this application, RuggedJ incurs no measurable overhead. We attribute this to our policy of rewriting only those objects that must be accessed remotely.

We observe a small performance decrease for this benchmark at 48 threads. This is the point at which the overhead of copying data for an additional four threads outweighs the benefit seen by their work. The point at which this occurs depends on the size of the workload; the overhead of copying is relatively constant, while the benefit of additional threads increases with the amount of data.

5.3.2 Molecular Dynamics

The second Java Grande application that we studied was MolDyn, a molecular dynamics simulator. This application models an *n*-body system where forces from each body act upon all others in the system. The simulation involves a pairwise resolution of forces that reduces to a single force vector on each particle. This calculation is iterative as each particle is affected by the total forces of the system.

Application Overview

As in the case of the Monte Carlo benchmark, Java Grande includes two versions of the MolDyn application: one using shared memory, the other using explicit MPJ. We created a hybrid application, using elements of each. Due to the complexity of the changes that we made to the data structures we rewrote much of the application. However, we left the core computations untouched and verified that our rewritten version produced the same result as the original.

This large-scale modification of the data structures for MolDyn was a more involved process than for Monte Carlo. Where the Monte Carlo simulation created temporary objects to perform calculations and return results, the MolDyn application stored all intermediate results in a central static array. This array could not be replicated as it was not immutable.

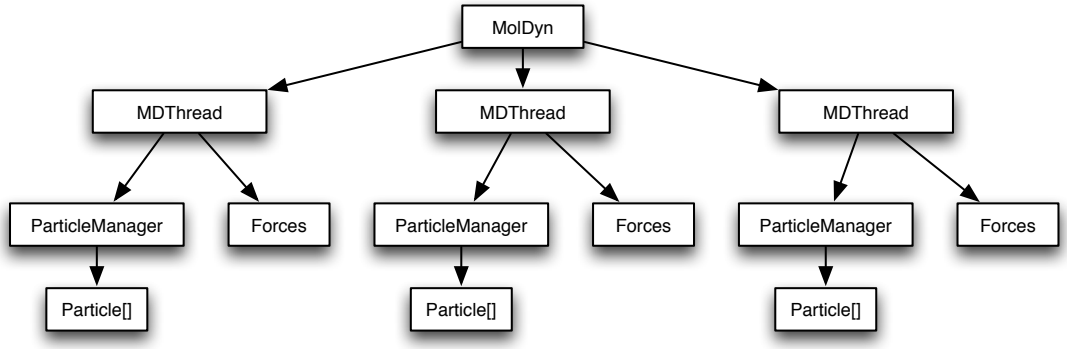


Figure 5.3.: MolDyn application structure

However each thread wrote to a different part of the array, meaning that they did not read or overwrite one another's results. Clearly this array was a bottleneck, so we modified the application to store intermediate results in a local, non-static array. As the results were only used by the thread that created them, this did not cause any change in the algorithm's correctness.

As with the Monte Carlo benchmark, we scaled the workload to run in approximately an hour on four threads. We did this by increasing the number of iterations for the system, and so the number of calculations.

Partitioning

Our rewritten version of the MolDyn benchmark is structured as shown in Figure 5.3. The eponymous main class of the application is `MolDyn`, which creates a number of worker `MDThread` objects, each of which encapsulates a thread. As with Monte Carlo, the threads are the decomposition point for distribution units. Each thread creates a `ParticleManager` object which generates the `Particle` objects under that thread's control. At the end of every iteration, each `MDThread` passes an array of `Forces` objects to the `MolDyn` object, which are merged with those from all other threads, and used as a starting point for the next iteration. Thus the particles themselves are never passed between threads, only the forces that they generate.

The `Forces` objects passed between threads contain three large arrays of **`double`** values, representing the *x*, *y* and *z* components of force vectors. As the computation of each iteration, as well as the resolution of forces, would require iteration over these arrays, it was essential that they be Direct. Iterating over a remote array requires repeated remote invocations, with the associated performance hit. We therefore declared the `Forces` class to be immutable, and rewrote any code that would have mutated a `Forces` object to instead create a new instance. This resulted in a slightly increased garbage collection load, but allowed `Forces` objects to be replicated and so operated upon locally.

The bulk of computation in the MolDyn benchmark is performed in the `ParticleManager` and `Particle` classes. We declare these classes to be System Direct, removing the overhead incurred by our object model. While these classes are mutable, they are not referenced from an external distribution unit, and so will never be replicated.

Performance Evaluation

Figure 5.4 shows the execution times for our rewritten Molecular Dynamics benchmark. We see similar scaling in this benchmark as in the Monte Carlo application, with a similar point at 48 threads where the communication cost begins to overwhelm the benefit of distribution.

Of interest in Figure 5.4 is a performance improvement when running RuggedJ with four threads over the untransformed version. We believe that this is an effect of the NUMA configuration of our test machines. We do not control which cores our application is assigned to, and so with a small number of threads the likelihood that data is in a distant memory bank increases. RuggedJ allocates several additional, sometimes large, data structures. By using more memory, the application data is distributed across more memory banks, leading to more reliable memory accesses.

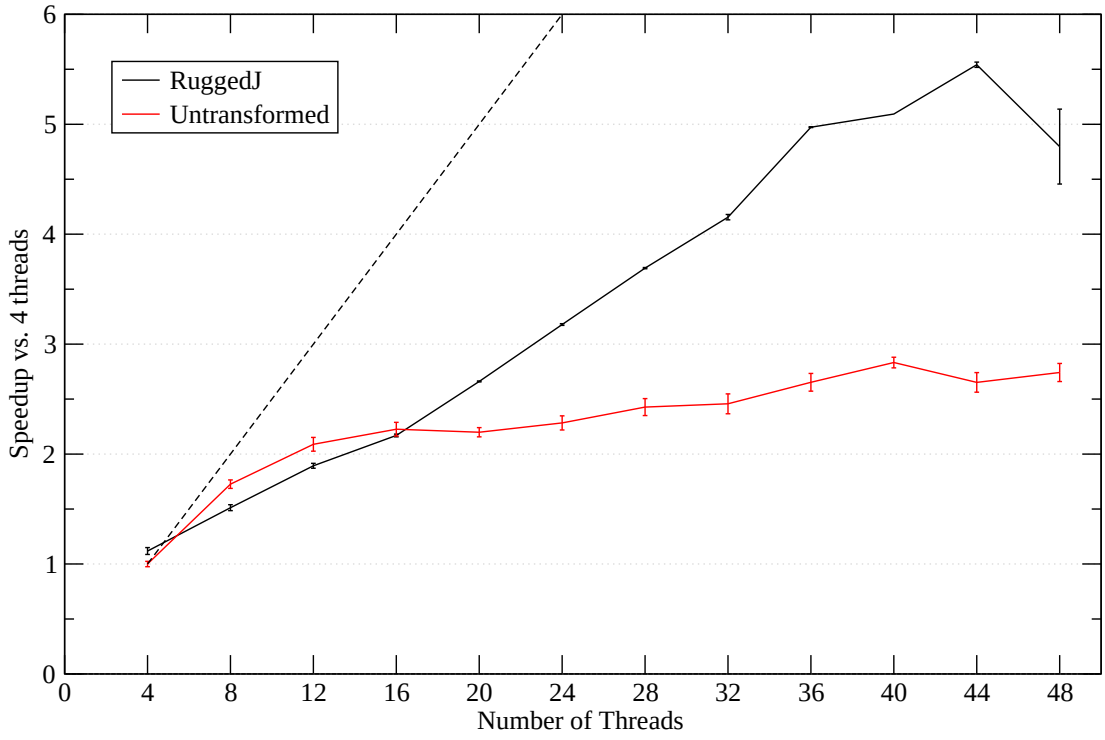


Figure 5.4.: MolDyn speedup (normalized to untransformed)

5.3.3 DNA Database Matching

Geneticists encode the DNA of plants and animals as character strings, with each character representing a particular amino acid. The DNA Database matching application takes as its inputs a "database" file of these DNA fragments (each of which is generally on the order of 1,000 characters), and a second file containing "query" fragments. The application performs string matching on the DNA sequences, finding the best matches (those with the smallest distance between the strings) from the database for each query. The matching algorithms used have a high space and time complexity (each string comparison is $O(mn)$, where m and n are the lengths of each string), and the matching application compares each database string with each query string.

Application Overview

The original DNA matching application (DSEARCH) was written to use a distributed Java framework developed by the Department of Computer Science at the National University of Ireland Maynooth [Keane et al., 2005]. This distribution system aims to provide horizontal scaling for applications, but uses a different approach from RuggedJ. Rather than our transparent system that provides the illusion of shared memory, the Maynooth researchers built an infrastructure that simplifies the explicit partitioning and distribution of work units, and executes applications as plug-ins to this framework. DSEARCH can thus be thought of to consist of three parts: the distribution framework, the plug-in component that partitions the workload, and the back-end logic that performs the matching algorithms.

Of these three components, the third part (the back-end logic) is most interesting to us. We extracted that part from the application and wrote our own driver program to partition the work load and initiate the matching algorithm. Discounting the distribution framework (which can be though of as performing a similar role to RuggedJ, although using a very different approach), we believe that our shared-memory abstraction allows for a simpler driver application. Our implementation was more concise (422 vs. 733 lines), and the DSEARCH driver had to conform to the many rules of the infrastructure, such as file naming conventions.

Partitioning

The structure of the DNA database matching application is shown in Figure 5.5. The search is launched by the `SearchCoordinator` class, which spawns multiple `ComparisonThread` instances. The database is divided up between the `ComparisonThreads`, which calculate the distances between the sequences in each thread's assigned portion of the database and each of the query sequences. The string matching is performed in the `neobio` package, a biocomputing library. The natural decomposition for this application is at the `ComparisonThread`.

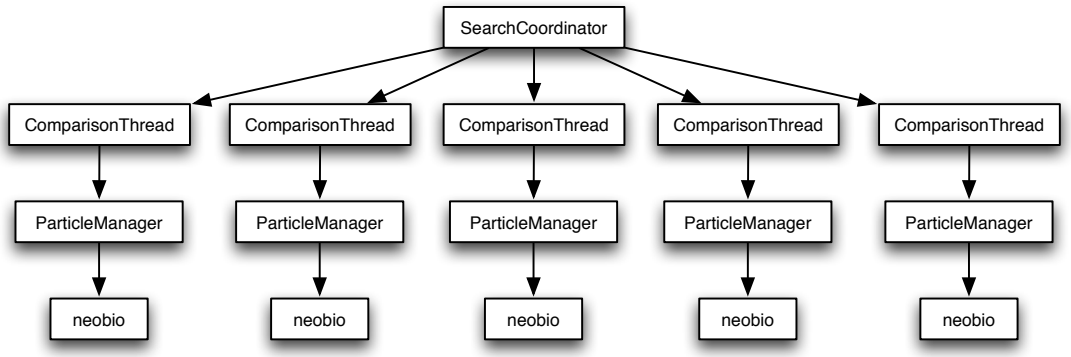


Figure 5.5.: DNA database matching application structure

The input database and query files are read by the `SearchCoordinator`, with each DNA sequence represented by a `Sequence` object. The database is divided into equally-sized arrays of `Sequences`, with one assigned to each `ComparisonThread`. The queries are represented by a single `Sequence` array, a reference to which is passed to each thread. We declare both `Sequence` and `Sequence[]` to be immutable, since they are constant after the initial database population step, allowing them to be replicated on each node.

The output of the DNA database matching program is a series of files, one for each query sequence, that lists the top `count` database matches (with `count` specified in a configuration file). Thus we must not only record the distance between each pair of sequences, we must also output a `String` that represents the transformations necessary to go from one sequence to the other. These matching `Strings` can be large (depending on the number of transformations and the length of the original sequences). Storing each of these transformations for the full $n \times m$ possible combinations of sequences is unfeasible; even on fairly modest database sizes the space requirements quickly exceed the memory available. Instead, we compute only the distance scores for each match, and then as a final act each `ComparisonThread` recalculates the matching for its top `count` matches for each query, storing the `String` representation for each. The results from each thread are encapsulated in Direct `ResultSet` instances, which are collated by the `SearchCoordinator`.

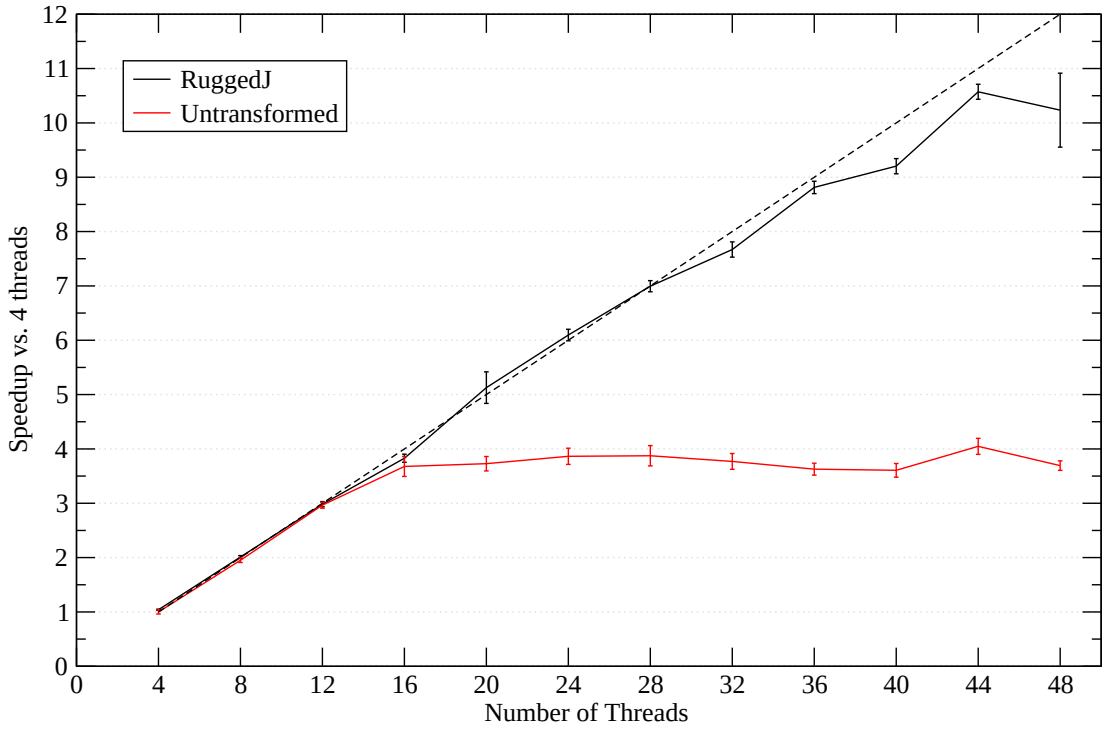


Figure 5.6.: DNA database matching speedup (normalized to untransformed)

Performance Evaluation

Figure 5.6 shows the performance of the DNA database application. Once again we see that RuggedJ incurs no measurable overhead for this application. Most computation occurs within the local computation library; by declaring this library to be Direct we minimized the overhead from our transformations. The small data set and intensive computational power required for this benchmark mean that we do not see a point where the cost of copying data overwhelms the benefit of additional processors on our cluster. In this case, however, we do see a decrease in scaling at 48 threads due to saturation of the available cores; remote accesses increase the number of active threads on each node leading to increased thread context switching.

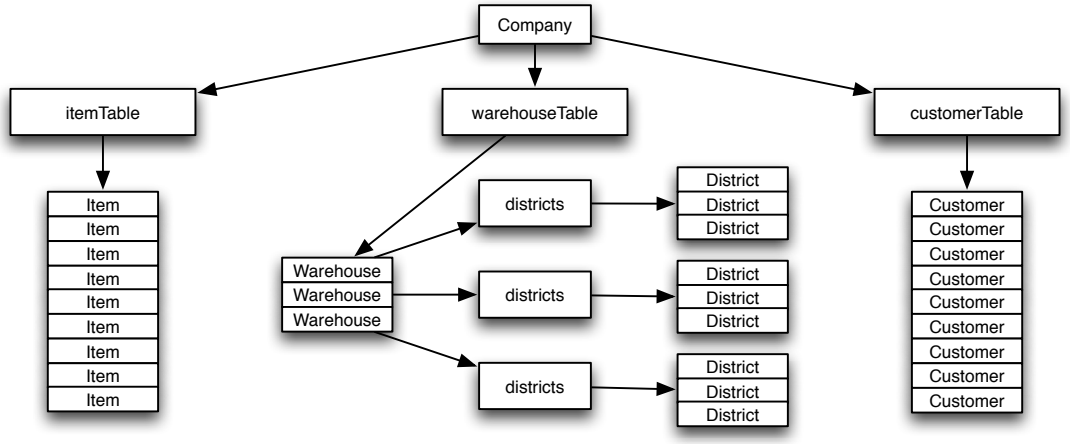


Figure 5.7.: SPECjbb2005's main database structure

5.3.4 SPECjbb2005

SPECjbb2005 [SPECjbb2005, 2005] is a standard Java benchmark application that implements the TPC-C workload [TPC], simulating the workflow of a company. The benchmark creates an in-memory database that tracks the warehouses and districts that make up the organization, as well as the customers and inventory managed by each warehouse. The benchmark then executes a number of different transaction types against the database. It creates and tracks orders, deliveries and payments, and produces reports on customers and stock levels. Each transaction is run on behalf of a warehouse, and interacts with the database. Performance is measured by transactional throughput within a fixed time.

Application Overview

SPECjbb2005 is designed to be parallel and scalable. Each warehouse has an associated thread that executes transactions; by increasing the number of warehouses (and therefore threads) the system as a whole can execute more transactions. Unfortunately, while the application is scalable it is not distributable. Figure 5.7 shows SPECjbb2005's major data structure. The `Company` object contains three major arrays: an `itemTable` that contains

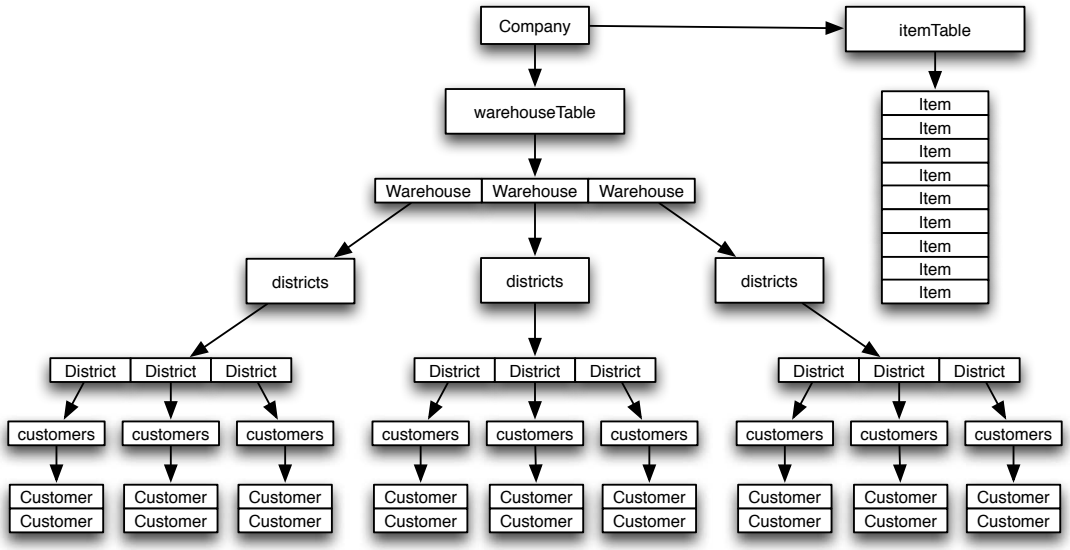


Figure 5.8.: Rewritten SPECjbb2005 application's main database structure

details of the products available in the company, a `warehouseTable` that holds references to the company's warehouses and a `customerTable` that lists the company's customers. Each warehouse holds an array of `District` objects. Elements of this data structure are referenced exclusively by ID (a **`short`** or **`int`** value that corresponds to an entry in the appropriate table). Thus a given `Customer` requires three indices to reference: the customer ID, the `District` to which it belongs, and the `Warehouse` containing that `District`. Obtaining a reference to that `Customer` means accessing the `Company` to get a `Warehouse` reference, the `Warehouse` to get the `District` reference, and finally the `District` to get the `Customer`. And since `Customer` references are not passed between methods, this lookup process is required every time a given `Customer` is referred to. This makes the `Company` object a clear bottleneck: every `Warehouse`, `District` or `Customer` reference must first query the `Company`.

We restructured the benchmark as shown in Figure 5.8. Since each `Customer` object is bound to a particular `District`, we distributed the `customerTable` structure between the `District` instances, removing the need to indirect through the `Company`. More importantly, we rewrote the application to refer to instances directly, rather than using IDs. Thus we pass

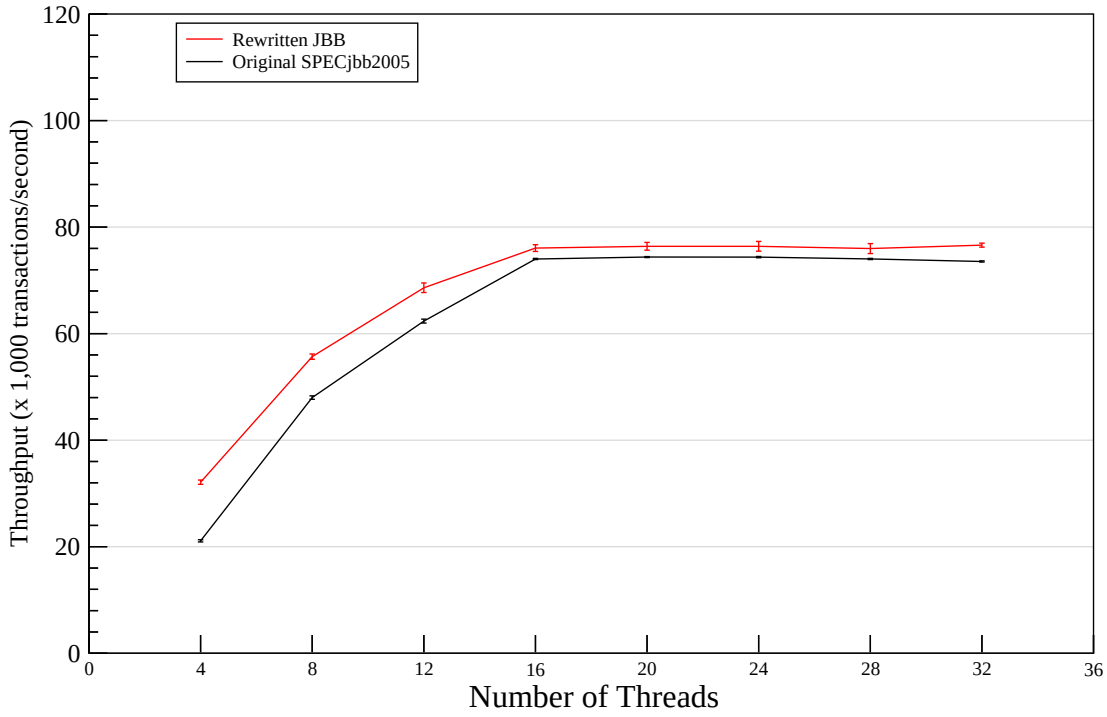


Figure 5.9.: Comparing the original SPECjbb2005 and rewritten JBB benchmarks

references rather than IDs, eliminating the vast majority of lookup operations. We modified the `Item` class slightly to allow it to be declared Direct, and so replicated on each node.

It should be noted that the changes we have made to the SPECjbb2005 benchmark alter its run-time characteristics in a variety of ways. For example, the new version of the application touches the `Company` object far less often, leading to different caching behavior. Similarly, by passing references rather than IDs we eliminate many redundant lookup operations. This major restructuring affects the benchmark in a more profound manner than the modifications that we performed on the Java Grande and DNA database applications, in which the data structures changed slightly but the core computation remained the same. While Figure 5.9 shows that the general scaling properties of the SPECjbb2005 are preserved in our rewritten version, and our application performs the same workload as the original (with a small speedup due to fewer lookups), we do not claim to provide a fully

accurate representation of SPECjbb2005. The remainder of this section discusses only our rewritten version of the benchmark (JBB), without reference to the original.

Partitioning

We define distribution units within JBB to be the threads associated with each warehouse. We collocate these threads with the warehouse (and associated `District` and `Customer` objects), ensuring that the majority of transactions performed within a distribution unit are local. The TPC-C specification upon which JBB is based requires that a minority of transactions are performed upon non-local warehouses. This necessitates that some transactions involve remote invocations. We parameterized our implementation to allow us to vary the percentage of remote accesses. As we will see when discussing the application performance, such accesses can cause significant performance degradation.

Another consequence of the requirement for non-local transactions is that any object in the benchmark's main data structure can be remotely referenced. This means that the `Company`, `Warehouse`, `District` and `Customer` classes must all implement the RuggedJ object model (none of these classes are immutable). Not only does this necessitate rewriting these classes, but it also means that the `Transaction` objects (which perform the bulk of the work in the application) must refer to transformed objects, and so cannot be declared System Direct. This affects performance, as all accesses must be indirected.

A final interesting feature of JBB's partitioning concerns the initial setup of the data structure. All `Company`, `Warehouse`, `District` and `Customer` instances are initialized using a data generation framework that creates random entries. This generator uses a single seeded `Random` instance that creates a repeatable set of values. We generate our data on RuggedJ's head node during the setup phase. Since the various data storage classes are mutable, we cannot simply replicate them across the network. Instead we migrate them at the end of the creation phase, allowing us to generate them using a globally unique `Random` instance, but place them on the nodes where they will be referenced.

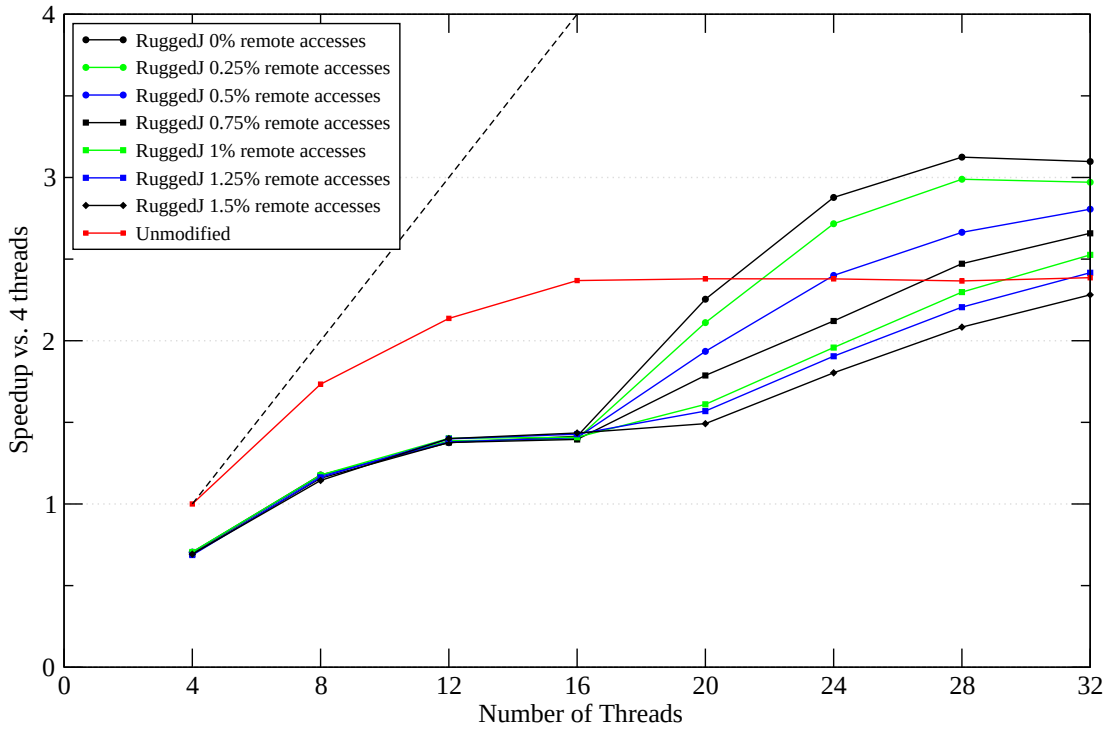


Figure 5.10.: Re-implemented version of SPECjbb2005 performance

Performance Evaluation

Figure 5.10 shows the performance of this application. JBB reports throughput over a fixed timing period (we measure for 120 seconds), and we report the result as transactions per second. The graph shows the performance of an untransformed version of this application, and the throughput on RuggedJ when we vary the fraction of remote accesses. We can see that RuggedJ incurs a significant overhead when running on a single machine. This is due to the highly-interconnected nature of the data structure and the transactions. Since the company, warehouse, and district objects are distributed by RuggedJ, they must be rewritten. And since the transactions refer to these data structures, they cannot be declared Direct. Therefore we incur the performance penalties from RuggedJ's transformations on all data accesses in the system.

Figure 5.10 shows the importance of locality within RuggedJ (and, indeed, any distributed application). When no transactions operate upon remote warehouses, the RuggedJ

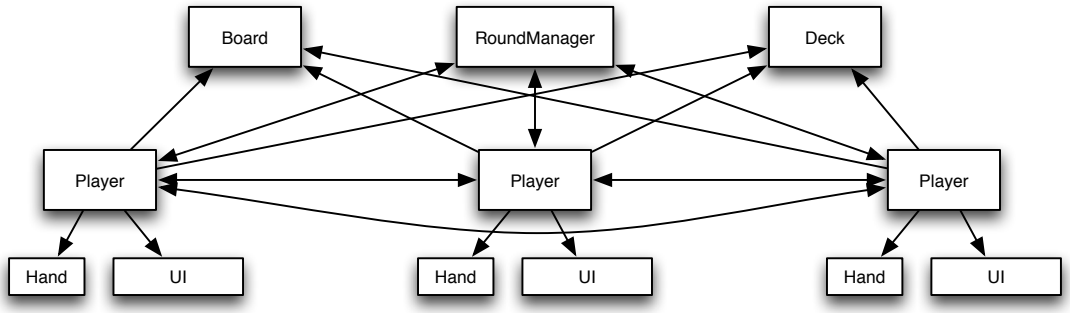


Figure 5.11.: Clue application structure

version significantly outperforms the untransformed version as we increase the number of warehouses. However, as the percentage of remote accesses increases, we see a steady decrease in the performance of the distributed version, until 1.5% of accesses are remote, at which point the distributed version only breaks even at 32 threads.

5.3.5 Clue

Our final benchmark was designed explicitly for RuggedJ. We implemented a multiplayer distributed version of the board game Clue in order to determine the benefit of developing applications from scratch with RuggedJ in mind. The run-time environment for this application differed significantly from the previous benchmarks: the game was deployed across the Internet, with multiple players running simultaneous interactive sessions. Thus performance was less of a concern, so long as the system remained responsive.

Application Overview

The system contains both client-server and peer-to-peer communication. This is a natural model for transparently distributed applications; in some cases objects communicate directly with one another without indirection through an intermediary controller, while global activities are performed by a central mediator. The application structure was designed as shown in Figure 5.11.

The key management unit in the application is the `RoundManager`. This creates a series of `Player` objects that interact with users; each user controls a `Player` through its associated `UI` (we show a single `UI` object to represent the classes that make up an interface in Swing). The `UI` is individual to the `Player`; `UI` objects do not communicate directly with one another. The `RoundManager` creates a `Board` and `Deck` that encapsulate the major data structures; the `Board` represents the space in which the game is played (including valid, invalid and special squares), while the `Deck` contains the cards that players use during the game. Each `Player` maintains a set of cards that make up his or her `Hand`, and refers to the `Board` and `Deck`.

A round in the game consists of each player in turn moving a counter around the board, and questioning one other player about the contents of their hand. All players observe both the movement and the question, while only the enquiring player knows which card is shown in response to a query; the other players simply see whether a card has been shown. We model this by a combination of one-to-one interactions and broadcast messages. The user locally updates his position, sends a query to another player (encoding the new position; the queried player may use this information in choosing the card to reveal), receives a response, and sends a summary of his turn to the `RoundManager`. The `RoundManager` broadcasts this summary to the remaining players, and indicates which player's turn is next.

Partitioning

The application as presented in Figure 5.11 appears at first to be too tightly coupled for distribution. However, our partitioning policy simplifies the structure, as shown in Figure 5.12. We declare the board and deck to be immutable; recall that the position of players is encoded in the broadcast turn summary from the `RoundManager` and so can be locally cached. Thus the natural distribution unit is the `Player`. This portioning ensures that the `UI` for each `Player` is entirely local to the distribution unit; the application has no shared UI state. This is desirable since Swing components interact closely with the underlying VM and operating system, and so have no meaning outside the context of the

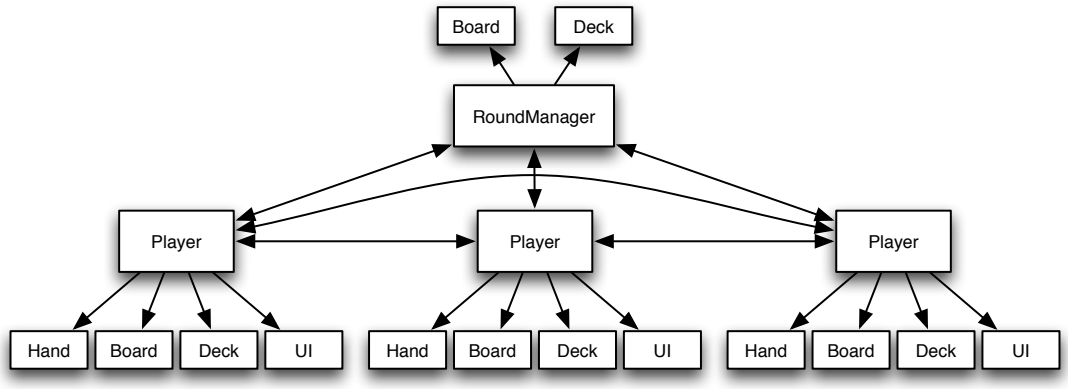


Figure 5.12.: Partitioning the Clue application

current node. Finally, we see that the only connections between distribution units are the necessary communication channels between individual players and between each player and the `RoundManager`. This minimizes communication to that which is necessary to the application.

We do not present performance numbers for the Clue application as any overheads introduced by RuggedJ are overwhelmed by the time taken for human players to move. However the Clue application shows how complex applications can be trivially distributed using RuggedJ, and how sensible partitioning choices can greatly simplify the structure of a distributed application.

5.4 Contributions

RuggedJ's partitioning interface strikes a balance between fine-grained developer control and transparent program development, while previous systems have emphasized one factor or the other. Distributed programming languages such as Emerald or X10 allow very precise control of object placement and movement, with explicit calls to reveal location information. At the other extreme, transparent distribution systems such as J-Orchestra or Addistant have imposed a class-based partitioning that is invisible to the developer. Each

system has its advantages; explicit object location allows the developer to tune the application partitioning, while static partitioning frees the developer from locality concerns.

RuggedJ allows both approaches, combining applications that were written with no explicit location information with an after-the-fact partitioning policy that affords precise control of object placement. This way, the majority of development can proceed using a familiar shared-memory single-machine model, while the developer retains the ability to fine-tune his partitioning strategy. Our partitioning plug-in system also allows developers to introspect on their applications and the network upon which they are running. By inserting partitioning call-backs a developer can invoke the partitioning policy at arbitrary points in execution, allowing migration decisions to be made at appropriate times. No other transparent distribution offers this level of control over partitioning.

5.5 Concluding Remarks

Distributable applications scale horizontally and can gain performance improvements by adding machines. Such applications can be decomposed into distribution units which can be allocated on remote machines, with minimal interaction between the threads and data of discrete units. Distributable applications can be further tailored to RuggedJ by following simple optimization rules such as maximizing immutability and designing with the RuggedJ object model in mind.

In this section we have examined the properties of applications that make the most of the RuggedJ transformation and run-time systems, and we have discussed the strategies by which distribution units can be allocated across a RuggedJ network. Finally, we have shown that our implementation of RuggedJ can handle large, realistic applications, and discussed how we tuned these applications to perform well under our system.

6 SUMMARY AND FUTURE WORK

Transparent distribution can allow distributable standard Java applications to execute across multiple machines with minimal programmer overhead. Transformed applications can show minimal performance degradation on a single host, while demonstrating significantly improved performance on a cluster.

6.1 Summary

In this dissertation we have presented RuggedJ, a specification-based transparent Java distribution framework. RuggedJ transforms standard Java applications to execute across a cluster of Java virtual machines with minimal developer input. A RuggedJ network is composed of an arbitrary number of heterogeneous machines, each running a compatible version of Java. Source applications are transformed using a rewriting class loader, and interact with a distributed run-time system. Applications are partitioned primarily by allocating instances of distributable units on the various nodes of the network, following a partitioning policy supplied by the developer.

Our rewriting class loader transforms the classes of an application to conform to the RuggedJ object model. We generate an interface that abstracts the class's protocol, which is implemented by three classes; one local, one remote and one proxy that holds a reference to the local or remote object, allowing for simple migration. We have developed a further set of transformations that allow us to integrate Java library code into our rewritten system, using four templates to transform these classes. We have specified the transformations required, as well as the classification algorithm that matches class to template.

Our run-time system manages execution of transformed code across the network. It tracks remote objects, replicates immutable state and migrates objects when necessary. We

have implemented the run-time system to support Java's semantics; we maintain globally unique static data, preserve object identity across multiple nodes, implement thread affinity and support the Java's monitor-based synchronization. The RuggedJ run-time system also provides a partitioning plug-in interface, to which application developers can create partitioning policies. These policies can introspect on the nodes of the network, allowing partitioning strategies to be tuned to the cluster upon which an application is executing.

We have also discussed the types of applications that perform well in a distributed environment. We have identified those features that lend an application to distribution in general, such as a decomposable structure with few serial sections, limited reliance on global data and a high level of immutability. We also discuss those qualities that cause an application to perform well under RuggedJ, including a simple inheritance hierarchy, separating performance-critical sections from remotely-accessible classes, and limiting use of certain language features. Finally, we discussed the implementations of several large, realistic applications, outlining the techniques that we used to partition them and demonstrating their performance on a RuggedJ network.

6.2 Future Work

There are several avenues of research that could follow from this work:

Reliability. As RuggedJ is deployed across larger networks and with longer-running applications, the likelihood of node failures increases. An interesting line of research would be to determine the level to which fault tolerance could be built into the system, whether through replication of work or by distributed transactions. Additionally, the object model and transformation techniques that we have outlined could allow data replication upon a single node, allowing for the possibility of research in transient failure models.

Changing networks. RuggedJ is currently targeted towards arbitrary network configurations. Our partitioning plug-in system allows developers to reason in terms of abstract resources rather than partitioning across a concrete network. However, this flexibility

exists only for the static network configuration; we cannot currently grow or shrink networks dynamically, perhaps in response to varying work loads or available hardware.

Reflection. RuggedJ's handling of reflection is presently on an ad-hoc basis, with our run-time system attempting to integrate reflective code into our rewritten classes. This approach is limited in the long term, and could be replaced by a more formal set of semantics that allow developers to use a strictly-defined subset of reflective behavior.

Caching. In this work, we have discussed the replication of data only in terms of immutable content. It would be possible to replicate mutable data in the system, so long as the replicas remained globally consistent. Adding a coherence mechanism to RuggedJ's run-time would allow this replication, and could offer significant performance improvements, relaxing some of the requirements set out in Chapter 5.

Java Memory Model. The Java Memory Model [Manson et al., 2005] provides a relaxed consistency model in which updates need not be propagated immediately; rather data values are guaranteed to be consistent only at synchronization points. We could make use of this consistency model to cache local modifications to data, updating the canonical version only when necessary.

Optimizations. There is room for some optimizations in our implementation. The major bottleneck in our rewriting system is the indirection required when obtaining local values; rather than using a `GetField` bytecode, we instead call a `get` method to obtain a field. This can lead to a major performance degradation, particularly when iterating across a large array. By implementing a static analysis or through programmer input we could determine methods that use purely local objects and bypass indirection in these instances. Another major bottleneck is large data structures, particularly arrays, which must be allocated on a single node and so remotely referenced by all others. We could break large arrays into smaller "arraylets", increasing the distributability of such data structures.

6.3 Conclusion

We have discussed the design and implementation of a prototype transparent Java distribution infrastructure. We have shown how the overhead imposed by our rewriting system can be minimized using a variety of techniques, including selectively omitting rewrites on performance critical sections. We have demonstrated that this system can distribute several realistic applications, and have shown that these applications running on a cluster exhibit scalability beyond that available to a single machine.

LIST OF REFERENCES

LIST OF REFERENCES

S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 229–240, 2007. 22

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 1967 Spring Joint Computer Conference*, pages 483–485, 1967. 99

D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004. 108

Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 4–11, 1999. 20

Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. Transparently obtaining scalability for Java applications on a cluster. *Journal of Parallel and Distributed Computing*, 60(10):1159–1193, 2000. 20

M. Austermann, P. Costanza, G. Kniesel, and H. Koch. The JMangler project. URL `http://roots.iai.uni-bonn.de/research/jmangler/`. 29

M. Baker and B. Carpenter. MPJ: A proposed Java message passing API and environment for high performance computing. In *The IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops, LNCS 1800*, pages 552–559, 2000. 69

A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of the 1st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 78–86, 1986. 21

A. P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the 10th ACM symposium on Operating Systems Principles (SOSP)*, pages 181–193, 1985. 21

A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy. The development of the Emerald programming language. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL)*, pages 11–1–11–51, 2007. 21

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006. 61

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Notices*, 30(8):207–216, 1995. 22

B. Bokowski and A. Spiegel. Barat—a front-end for Java. Technical Report B-98-09, Freie Universität Berlin, Sept. 1998. 30

E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, 2002. 30

M. Busch. Adding dynamic object migration to the distributing compiler Pangaea. Master's thesis, FU Berlin, FB Mathematik und Informatik, 2001. 27

D. Caromel and J. Vayssière. A Java framework for seamless sequential, multi-threaded, and distributed programming. In *The Workshop on Java for High-Performance Network Computing*, 1998. 24

D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency—Practice and Experience*, 10(11–13):1043–1061, 1998. 24

P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Notices*, 40(10):519–538, 2005. 22

X. Chen and V. H. Allan. MultiJav: A distributed shared memory system based on multiple Java virtual machines. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 91–98, 1998. 20

S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, pages 313–336, 2000. 29

S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE)*, pages 364–376, 2003. 29

A. F. da Silva, M. Lobosco, and C. L. de Amorim. An evaluation of cJava system architecture. In *Symposium on Computer Architecture and High Performance Computing*, page 91, 2003. 20

M. Dahm. Doorastha—a step towards distribution transparency. In *JIT, 2000*, 2000a. 24

M. Dahm. The Doorastha system. Technical Report B-1-2000, Freie Universität Berlin, 2000b. 24

M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin, 2001. 30

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 106

P. Eugster. Uniform proxies for Java. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 139–152, 2006. 26

M. Factor, A. Schuster, and K. Shagin. Instrumentation of standard libraries in object-oriented languages: The Twin Class Hierarchy approach. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 288–300, 2004. 26, 52

E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Mar. 1995. 98

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005. 80

D. Hagimont and D. Louvegnies. Javanaise: Distributed shared objects for Internet cooperative applications. In *Proceedings of Middleware'98*, 1998. 24

S. S. Huang and Y. Smaragdakis. Easy language extension with Meta-AspectJ. In *Proceedings of the 28th ACM International Conference on Software Engineering (ICSE)*, pages 865–868, 2006. 29

L. Iftode. *Home-Based Shared Virtual Memory*. PhD thesis, Princeton University, 1998. 19

E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988. 22

T. M. Keane and T. J. Naughton. DSEARCH: Sensitive database searching using distributed computing. *Bioinformatics*, 21(8):1705–1706, 2005. 61, 107

T. M. Keane, A. J. Page, J. O. McInerney, and T. J. Naughton. A high-throughput bioinformatics distributed computing platform. In *Proceedings of the 18th IEEE International Symposium on Computer-Based Medical Systems (CBMS)*, pages 377–382, 2005. 116

P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, 1994. 20

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, 1997. 29

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001a. 29

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44:59–65, 2001b. 29

G. Kniesel, P. Costanza, and M. Austermann. JMangler—a framework for load-time transformation of Java class files. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 100 – 110, 2001. 29

P. Launay and J.-L. Pazat. A framework for parallel programming in Java. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe)*, pages 628–637, 1998a. 23

P. Launay and J.-L. Pazat. Generation of distributed parallel Java programs. Technical Report PI-1171, Institut de Recherche en Informatique et Systemes Aleatoires, 1998b. 23

S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 36–44, 1998. 32, 44

T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice-Hall, 2nd edition, 1999. 85

N. Liogkas, B. MacIntyre, E. D. Mynatt, Y. Smaragdakis, E. Tilevich, and S. Voida. Automatic partitioning: A promising approach to prototyping ubiquitous computing applications. *IEEE Pervasive Computing*, 3(3):40–47, 2004. 14

B. Liskov, D. Curtis, P. Johnson, and R. Scheifer. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 111–122, 1987. 21

M. Lobosco. A new distributed JVM for cluster computing. In *Proceedings of the 9th International Euro-Par Conference*, 2003. 20

M. Lobosco, O. Loques, and C. L. de Amorim. Reducing memory sharing overheads in distributed JVMs. In *Proceedings of the 1st International Conference on High Performance Computing and Communications (HPCC)*, pages 629–639, 2005. 20

J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 378–391, 2005. 130

J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 72–80, 1999. 62

P. McGachey, A. L. Hosking, and J. E. B. Moss. Pervasive load-time transformation for transparently distributed Java. *Electronic Notes in Theoretical Computer Science*, 253(1): 47–64, 2009a. 7

P. McGachey, A. L. Hosking, and J. E. B. Moss. Classifying Java class transformations for pervasive virtualized access. In *Proceedings of the 8th International Conference on. Generative Programming and Component Engineering (GPCE)*, pages 75–84, 2009b. 79

A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu. Towards a distributed platform for resource-constrained devices. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 43–51, 2002. 19

G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), Apr. 1965. 95

OMG. The Common Object Request Broker: Architecture and Specification. Technical Report 91.12.1 rev 1.1, Object Management Group, 1992. 25

M. Philippsen and B. Haumacher. Locality optimization in JavaParty by means of static type analysis. *Concurrency—Practice and Experience*, 12(8):613–628, July 2000. 23

M. Philippsen and M. Zenger. JavaParty—transparent remote objects in Java. *Concurrency—Practice and Experience*, 9(11):1225–1242, Nov. 1997. 23

M. Robinson and P. Vorobiev. *Swing Second Edition.* Mannings Publications, 2003. 108

D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proceedings of the 25th IEEE/ACM International Conference Automated Software Engineering (ASE)*, pages 114–123, 2005. 26

R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, 1998. 19

SPECjbb2005. Java server benchmark, 2005. URL `http://www.spec.org/jbb2005`. Standard Performance Evaluation Corporation. 61, 119

SPECjvm98. Java virtual machine benchmarks, 2008. URL `http://www.spec.org/jvm2008/`. Standard Performance Evaluation Corporation. 61

A. Spiegel. Automatic distribution in Pangaea. In *Proceedings of the 3rd International Workshop on Communications-Based Systems (CBS)*, pages 119–146, 2000. 27

A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, Freie Universität Berlin, Dec. 2002. 27

A. Spiegel. Pangaea: An automatic distribution front-end for Java. In *Proceedings of the IPPS/SPDP Workshops*, pages 93–99, 1999. 27

G. L. Steele, Jr. Parallel programming and code selection in Fortress. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–1, 2006. 22

F. Steimann. The Infer Type refactoring and its use for interface-based programming. *Journal of Object Technology*, 6(2), 2007. 25

Sun Microsystems, Inc. Java remote method invocation specification, a. URL `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`. 23

Sun Microsystems, Inc. Dynamic proxy classes, b. URL `http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html`. 25

Sun Microsystems, Inc. The JVM tool interface, c. URL `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti`. 46

E. Tanter, M. Ségura-Devillechaise, J. Noyé, and J. Piquer. Altering Java semantics via bytecode manipulation. In *Proceedings of the 1st International Conference on Generative Programming and Component Engineering (GPCE)*, pages 283–298, 2002. 29

M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of “legacy” Java software. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, 2001. 18

Terracotta Inc. URL `http://terracotta.org`. 17

The Apache Software Foundation. Hadoop. URL `http://hadoop.apache.org/`. 106

The Java Grande Forum. The Java Grande benchmark suite. URL `http://www.epcc.ed.ac.uk/research/java-grande`. 62, 107

E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, pages 178–204, 2002. 14, 15

E. Tilevich and Y. Smaragdakis. Portable and efficient distributed threads for Java. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference*, pages 478–492, 2004. 16, 82

E. Tilevich and Y. Smaragdakis. Transparent program transformations in the presence of opaque code. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 89–94, 2006. 15, 58

E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Transactions on Software Engineering and Methodology*, 19(1):1–40, 2009. 14

E. Tilevich, Y. Smaragdakis, and M. Handte. Appletizing: Running legacy Java code remotely from a Web browser. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 91–100, 2005. 14

TPC. The TPC benchmarks. URL `http://www.tpc.org/`. 119

W. Yu and A. L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency—Practice and Experience*, 9(11):1213–1224, 1997. 20

VITA

VITA

Phil McGachey was born in Glasgow, Scotland, attending Trinity High School and St. Aloysius College. He earned a BSc in Software Engineering at Glasgow University, graduating with a First Class Honors degree in 2002. Phil spent the second year of his undergraduate course at Boston College in Chestnut Hill, Massachusetts.

On graduating from Glasgow, Phil moved to Purdue University to work with Prof. Tony Hosking in the Secure Software Systems lab. He completed summer internships at Sun Microsystems in 2005 and at Intel's Programming Systems Lab in 2006 and 2007. He completed his MS degree in 2005 and his PhD in 2010.

Phil's research interests have centered around run-time systems, spending time working on Java VMs, garbage collection and transparent distribution. Outside of work, he enjoys golf, traveling, model ship building, and the acquisition of tiny power tools.